PROYECTO FIN DE GRADO

TÍTULO: VISUALIZADOR DE PROPAGACIÓN DE ONDAS EN TIEMPO REAL

AUTOR/A: DAVID ARAGONÉS MALLÉN

TITULACIÓN: SONIDO E IMAGEN

TUTOR/A: MARÍA PILAR OCHOA PÉREZ

DEPARTAMENTO: ELECTRÓNICA FÍSICA, INGENIERÍA ELÉCTRICA Y FÍSICA APLICADA

VºBº TUTOR/A

Miembros del Tribunal Calificador:

PRESIDENTE/A: ELENA BLANCO MARTÍN

TUTOR/A: MARÍA PILAR OCHOA PÉREZ

SECRETARIO/A: MANUEL VÁZQUEZ LÓPEZ

Fecha de lectura: 26 de Julio de 2023

Calificación:

El Secretario/La Secretaria,

Agradecimientos

A mis padres. Un enorme, gigantérrimo e infinito **GRACIAS**. Todo lo que soy, todo lo que he sido y todo lo que seré, se lo debo a ellos. Ellos han sido mi hogar, mi espada, mi escudo, mi faro, mi oasis y mi ejemplo. No existen ni existirán palabras en ningún idioma de la historia para expresar lo afortunado que soy de ser su hijo. Mamá, Papá, podéis estar orgullosos de ser los mejores padres del multiverso. Os quiero. Una vez más: **GRACIAS**.

A mi familia, por estar ahí siempre para apoyarme en todos los sentidos.

A mis amigos. Por todo el apoyo, amor, comprensión y tolerancia que me han regalado todos estos años. Por escucharme, aconsejarme y, por qué no decirlo, abroncarme cuando era necesario. Gracias.

A mis compañeros de trabajo, por facilitarme la vida siempre que han podido para que pudiera terminar esta maldita y preciosa carrera estos últimos dos años. Por cubrirme cuando era necesario y dejarme tiempo para estudiar. Gracias.

A Pilar, mi tutora. LA tutora. Por creer en mí ciegamente para hacer este proyecto y volcarse para que fuera lo más perfecto posible. Gracias.

A Brianda. MI COMPAÑERA, mi mejor amiga, mi hogar y el amor de mi vida. La luz con la que bañas a los de tu alrededor podría iluminar hasta la más oscura de las singularidades. Has sido inspiración y fuerza para mí desde que te conozco. En cada pequeño detalle, en cada palabra, en cada gesto y en cada mirada. La palabra “vida” ha cobrado un sentido nuevo desde que llegaste a ella y, desde luego, este trabajo hubiera sido cien veces más difícil sin ti. Te quiero. GRACIAS.

Resumen

Visualizador de propagación de ondas en tiempo real

El objetivo principal del proyecto es diseñar y desarrollar una herramienta que permita visualizar el comportamiento de las ondas acústicas en un medio elástico (en este caso el aire) y, además, ofrezca la posibilidad de modificar los parámetros principales de la onda actualizando la representación en tiempo real. De esta manera se pretende que sirva como material de ayuda para la asignatura Propagación de Ondas, impartida en la Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunación.

Las ondas acústicas al propagarse en un medio interaccionan con las partículas que lo constituyen, de forma que éstas se desplazan respecto a sus posiciones de equilibrio. Es este desplazamiento de las partículas el que se va a representar y visualizar en tres dimensiones. Para que esta visualización tridimensional sea correcta, dichas partículas han de ser representadas en un volumen y, a su vez y con el mismo fin, ha de haber una cantidad considerable de partículas siendo excitadas a la vez en pantalla.

Renderizar y calcular la posición en tiempo real de tantos elementos puede suponer un gran problema de rendimiento para dispositivos con controladores gráficos de características limitadas.

Como tecnologías capaces de suplir el condicionante tecnológico anteriormente descrito se usan para el desarrollo la plataforma de desarrollo y creación de contenido en tiempo real Unity y la tecnología DOTS (*Data-Oriented Technology Stack*), desarrollada por Unity Technologies.

En Unity, para un renderizado eficiente, se hace uso del flujo de renderizado URP (*Universal Render Pipeline*). URP permite crear gráficos optimizados fácilmente, ya que está diseñado para ser compatible con todo tipo de dispositivos.

DOTS se basa en tres pilares: ECS (*Entity Component System*), el *Job System* y el *Burst Compiler*. Estos componentes trabajan juntos para optimizar el uso de la memoria, el procesamiento paralelo y la compilación de código nativo, pero se puede usar cualquiera de ellos por separado. En este proyecto se hace uso de ECS y del *Job System*.

La primera parte del proyecto se ha enfocado al diseño de una arquitectura sólida y reescalable, haciendo uso de manejadores o *managers* estáticos, que centralizan el comportamiento de las diferentes partes del código, desde el flujo de la aplicación, hasta el propio comportamiento de las partículas simuladas. Esta parte del proyecto también se ha dedicado a la investigación y aprendizaje de la tecnología DOTS, con el objetivo de poder evaluar qué herramientas de dicha tecnología serán útiles para el desarrollo de este proyecto.

La segunda parte se ha dedicado a la implementación de la aplicación diseñada, haciendo especial énfasis en la eficiencia de esta, así como al realismo a la hora de representar la propagación de la onda en el aire.

Por último, se ha trabajado en la experiencia de usuario, tratando de hacer la navegación por el espacio de simulación lo más cómoda e intuitiva posible.

Tras implementar las partículas, tanto con el sistema convencional de Unity, como con DOTS, se ha observado un incremento masivo del rendimiento, pudiendo visualizar hasta

ocho veces más partículas con la misma tasa de FPS (*Frames* por segundo) con este último.

También se ha podido comprobar la enorme utilidad que tiene Unity para realizar aplicaciones de simulación y visualización de manera cómoda y eficiente, gracias a todas sus herramientas gráficas y las facilidades que aporta para mejorar la experiencia de usuario.

Abstract

Real Time Wave Propagation Display

The main objective of the project is to design and develop a tool that allows visualizing the behavior of acoustic waves in an elastic medium (in this case air) and, in addition, offers the possibility of modifying the main parameters of the wave by updating the representation in real time. It is intended to serve as support material for the Wave Propagation course, taught at the Higher Technical School of Engineering and Telecommunication Systems.

As acoustic waves propagate in a medium, they interact with the particles that make it up, so that these particles move around their equilibrium positions. It is this displacement of the particles that will be represented and visualized in three dimensions. For this three-dimensional visualization to be correct, these particles must be represented in a volume and, at the same time and for the same purpose, there must be a considerable number of particles being excited at once on screen.

Rendering and calculating the position in real time of so many elements can pose a major performance problem for devices with limited graphics controllers.

As technologies capable of meeting the technological constraint described above, the development platform and real-time content creation Unity and the DOTS technology, developed by Unity Technologies, are used for development.

For more efficient rendering, the URP rendering flow is used in Unity. URP allows to easily create optimized graphics, as it is designed to be compatible with all kinds of devices.

DOTS is based on three pillars: ECS, Job System and Burst Compiler. These components work together to optimize memory usage, parallel processing, and native code compilation, but any of them can be used separately. ECS and Job System are used in this project.

The first part of the project has focused on designing a solid and scalable architecture, making use of static handlers or managers that centralize the behavior of different parts of the code, from application flow to the behavior of simulated particles. This part of the project has also been dedicated to researching and learning DOTS technology, with the aim of being able to evaluate which tools from this technology will be useful for developing this project.

The second part has been dedicated to implementing the designed application, with special emphasis on its efficiency as well as realism when representing wave propagation in air.

Finally, user experience has been a focus of work, trying to make navigation through the simulation space as comfortable and intuitive as possible.

After implementing particles using both Unity’s conventional system and DOTS, a massive increase in performance has been observed, being able to visualize up to eight times more particles with the same FPS rate with the latter.

It has also been possible to verify the enormous usefulness that Unity has for comfortably and efficiently carrying out simulation and visualization applications thanks to all its graphic tools and facilities it provides for improving user experience.

Índice de Contenido

Índice de Tablas y Figuras

Lista de Acrónimos

- BIRP (Built in Render Pipeline)
- C# (Csharp)
- CLR (Common Language Runtime)
- DOTS (Data-Oriented Technology Stack)
- ECS (Entity Component System)3
- FPS (Frames por segundo)
- HDRP (High-Definition Render Pipeline)
- HID (Human Interface Device)
- HLSL (High Level Shader Language)
- IDE (Integrated Development Environment)
- LTS (Long Term Support)
- UI (User Interface)
- URP (Universal Render Pipeline)

1 Introducción

Este proyecto de fin de grado surge de la idea de crear una herramienta para facilitar, tanto a alumnos como a docentes la asignatura Propagación de Ondas, impartida en la Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunación, facilitando de manera visual el entendimiento y la impartición de la teoría vista en la asignatura.

Inicialmente se pretendía crear el simulador para todos los tipos de ondas vistos en la asignatura, pero finalmente se ha centrado en ondas acústicas planas, dejando una estructura sencilla para la extensión futura del proyecto, tanto para ondas acústicas esféricas, como ondas electromagnéticas.

El proyecto se centra en el estudio, simulación y visualización de las ondas sonoras, que son ondas elásticas longitudinales que requieren un medio material para su propagación.

Aunque en el lenguaje común el término sonido se asocia con la percepción auditiva humana, desde una perspectiva física no existe diferencia entre las señales que el oído humano puede percibir y aquellas que están fuera del rango audible.

La ciencia encargada del estudio de los métodos de producción, recepción, transmisión y absorción del sonido se conoce como Acústica y está estrechamente relacionada con diversas áreas de la ingeniería. Este trabajo se enfoca en la simulación de la propagación del sonido a través de un fluido, entendiendo por fluido cualquier sustancia en estado líquido o gaseoso. También se examina la relación entre las perturbaciones en el desplazamiento y la presión, visualizando el desplazamiento de las partículas (1).

Para realizar dicha simulación se utiliza el motor de desarrollo gráfico Unity, que inicialmente era un motor de creación de videojuegos, pero su popularidad y utilidad se han extendido hasta aplicaciones, tanto de investigación, como de simulación, formación y entrenamiento.

Para que la aplicación pueda ser usada por la mayor cantidad de usuarios posible, se tienen en cuenta dos cosas:

- Unity permite crear aplicaciones multiplataforma para Windows, MacOS y Linux, pudiendo trabajar desde el mismo proyecto para esos tres sistemas operativos sin necesidad de hacer ningún tipo de *port*.
- Unity cuenta con una gran cantidad de herramientas que permiten, usadas adecuadamente, un gran ahorro de recursos en cualquier equipo, haciendo así la aplicación compatible con la mayoría de los ordenadores en uso en este momento, aunque sus características o componentes sean muy antiguos.

Una de las herramientas mencionadas es la tecnología DOTS, desarrollada por Unity Technologies, permite optimizar el procesamiento de grandes cantidades de objetos en una misma escena, repartiendo el trabajo entre los diferentes hilos del microprocesador.

A su vez se usa el flujo de renderizado URP, el cual es ideal para plataformas de bajo rendimiento, como ordenadores de gama baja o antiguos, dispositivos móviles o web.

Reducir el rendimiento requerido por la aplicación también es muy importante en el marco medioambiental y económico. Al reducir el rendimiento requerido se reduce la energía consumida, por lo que disminuyen las emisiones generadas y se abarata el coste de usar la aplicación.

La aplicación se ha creado con una arquitectura de máquina de estados, independizando la implementación de características en cada estado, como la interfaz de usuario, los elementos cargados en la escena o las acciones que puede realizar el usuario. Esto facilita también la creación de nuevos estados para poder implementar el resto de los tipos de ondas en el futuro de manera independiente y sin necesidad de tocar el sistema ya implementado.

La interfaz de usuario es adaptativa y reactiva, lo que quiere decir que se renderiza con la misma relación de tamaño en cualquier ventana y además se adapta en tiempo real, en caso de cambiar el tamaño de dicha ventana. A su vez también está automatizada, de manera que es muy sencillo implementar nuevos elementos, como características de onda o del medio sin apenas esfuerzo.

La arquitectura de la gestión de entradas de *Hardware* está centralizada en una misma fuente que traduce las acciones del usuario en datos apropiados para ser leídos desde cualquier parte de la aplicación. A su vez, gracias al *Input System* de Unity, es muy sencillo adaptar estos datos a otro tipo de entradas, como pantallas táctiles, mandos de consola e, incluso, controladores de gafas de realidad virtual.

A continuación, se detalla la estructura de la memoria, indicando brevemente el contenido de cada capítulo:

2. Marco tecnológico

   Se proporciona un contexto tecnológico y una base teórica sobre Unity y las herramientas utilizadas. Esta base teórica es necesaria para facilitar la comprensión de los apartados posteriores.

3. Especificaciones y restricciones de diseño

   Se describen las especificaciones de la solución desarrollada, así como sus limitaciones de diseño. La principal restricción de diseño que se plantea para el desarrollo de la aplicación es la capacidad de procesado y renderizado que pueden tener los dispositivos en los que se va a reproducir.

4. Descripción de la solución propuesta

   Se describen y razonan detalladamente, tanto las decisiones de diseño, como el desarrollo de todos los elementos que componen la aplicación desarrollada en este PFG.

5. Resultados

   Se proporciona un análisis de los resultados de rendimiento, comparando el flujo de trabajo, tanto con *GameObjects*, como con el sistema DOTS.

6. Planos

   Se describen de forma gráfica los tres sistemas principales que componen la aplicación: arquitectura, manejo de entradas y partículas.

7. Presupuesto

   Se desglosa el presupuesto requerido para la realización de este proyecto.

8. Manual de usuario

   Se proporciona un manual con instrucciones detalladas del uso adecuado de la aplicación, así como instrucciones para acceder al tutorial interactivo.

9. Impacto del proyecto

   Se describe el impacto social, económico y medioambiental del proyecto.

10. Conclusiones

    Se detallan las conclusiones extraídas tras la finalización del desarrollo del proyecto, así como propuestas de líneas futuras posibles para su mejora.

2 Marco tecnológico

La simulación visual de cualquier elemento, desde una infoarquitectura hasta la evolución de datos a lo largo del tiempo, es un campo que siempre requiere de una considerable cantidad de recursos visuales por parte del dispositivo que va a reproducir la simulación. Por otra parte, incluir el elemento de tiempo real añade un extra de estrés al rendimiento, por lo que se han de elegir las herramientas que se van a utilizar con cuidado para lograr el mejor equilibrio visualización-rendimiento posible.

2.1 Unity

Unity lanza al mercado su primera versión en el año 2005 (2). Nace como necesidad de una herramienta de creación de videojuegos para el sistema operativo Mac OS X, que carece en ese momento de herramientas sencillas para sus desarrolladores, al contrario que Microsoft para Windows. Poco a poco Unity va expandiendo su marco de trabajo, abarcando otros sistemas operativos de escritorio (Windows y sistemas basados en Linux), consolas (PlayStation, Xbox y Nintendo Switch, principalmente) y sistemas operativos móviles (Windows Phone OS, iOS y Android) y escalando mucho en popularidad en el desarrollo para éstos últimos. Hoy en día Unity ya no es únicamente un motor de videojuegos. Se define a sí mismo como “La plataforma de creación de contenido en tiempo real líder en el mundo” (3). Se usa para producciones audiovisuales, simulación, formación, exhibiciones virtuales e incluso investigación.

Para el desarrollo de *software* Unity utiliza el lenguaje de programación C# (*Csharp*).

2.2 Lenguaje C#

C# es un lenguaje de programación creado por Microsoft para su marco de trabajo .NET (4). Evoluciona de los lenguajes C y C++, siendo su gran diferencia el uso de memoria seguro mediante el CLR (*Common Language Runtime)* y su *Garbage Collector* (recolector de basura), que va liberando el espacio de memoria sin usar sin necesidad de que el desarrollador tenga que hacerlo específicamente.

2.3 Flujo de renderizado

Unity cuenta con tres flujos de renderizado

- BIRP (*Built in Render Pipeline*)
- HDRP (*High-Definition Render Pipeline*)
- URP

El BIRP de Unity es el sistema de renderizado predeterminado que utiliza el motor para dibujar las escenas. El BIRP ofrece un balance entre rendimiento y calidad, y permite personalizar algunos aspectos del proceso de renderizado mediante *scripts* o *shaders*. El BIRP se puede usar para una variedad de proyectos, desde juegos 2D hasta experiencias de realidad virtual.

Sin embargo, el BIRP tiene algunas limitaciones en cuanto a la iluminación, las sombras, los efectos visuales y el post-procesado. Para superar estas limitaciones, Unity ofrece dos

alternativas: el HDRP y el URP. Estos son sistemas de renderizado más modernos y optimizados que ofrecen más control y flexibilidad sobre el aspecto visual de las escenas.

El HDRP está diseñado para proyectos que requieren una calidad gráfica muy alta, como juegos AAA, simulaciones o películas. El HDRP utiliza técnicas avanzadas de iluminación, sombreado y post-procesado para crear imágenes realistas y detalladas. El HDRP también soporta el *ray tracing*, que permite simular la interacción de la luz con los objetos de forma más precisa. El HDRP es ideal para plataformas de alto rendimiento, como PC o consolas.

El URP está diseñado para proyectos que requieren un rendimiento óptimo, como juegos móviles, web o realidad aumentada. El URP utiliza técnicas simplificadas de iluminación, sombreado y post-procesado para crear imágenes estilizadas y eficientes. El URP también soporta el renderizado por *script*, que permite modificar el proceso de renderizado mediante código. El URP es ideal para plataformas de bajo rendimiento, como dispositivos móviles o web (5).

2.4 Gestión de entradas de Hardware

Unity cuenta con dos sistemas de gestión de entrada de *Hardware*:

- *Input Manager*
- *Input System*

El *Input Manager* de Unity es una herramienta que permite configurar los controles de entrada para diferentes plataformas y dispositivos. Con el *Input Manager*, se puede asignar un nombre a cada eje de entrada y definir qué teclas, botones o *joysticks* lo activan. También se puede ajustar la sensibilidad, la gravedad y el tipo de curva de cada eje.

El *Input Manager* es útil para crear juegos que se adapten a diferentes tipos de entrada, como teclado, ratón, mando o pantalla táctil. Sin embargo, tiene algunas limitaciones, como no poder detectar eventos de entrada individuales, no soportar dispositivos personalizados o no ofrecer una forma sencilla de cambiar entre diferentes esquemas de control (6).

Por eso, Unity ha desarrollado el *Input System*, un nuevo sistema de entrada que reemplaza al *Input Manager* y ofrece más flexibilidad y funcionalidad. El *Input System* permite crear acciones de entrada que se pueden asignar a diferentes dispositivos y controles, y que se pueden agrupar en perfiles o mapas según el contexto del juego. Además, el *Input System* soporta una gran variedad de dispositivos, incluyendo los que usan el protocolo HID (*Human Interface Device*), y permite acceder a datos de entrada más detallados, como la presión o la vibración (7).

2.5 DOTS

La tecnología DOTS desarrollada por Unity Technologies, permite optimizar el procesamiento de grandes cantidades de objetos en una misma escena, repartiendo el trabajo entre los diferentes hilos del microprocesador (8). Dicho reparto no sólo permite tiempos de procesado mucho más rápidos, si no que, en dispositivos móviles, mejora sustancialmente el tiempo de duración de la batería, así como el control de temperatura del dispositivo. DOTS, a fecha de escritura de este documento, aún no ha alcanzado su fase de mercado, pero su desarrollo ha avanzado lo suficiente como para ser de gran utilidad. La finalidad de DOTS es ideal para uno de los objetivos del proyecto, ya que permite la

simulación de grandes cantidades de partículas en un volumen siendo excitadas por una onda mecánica.

DOTS se basa en tres pilares: ECS, el *Job System* y el *Burst Compiler*. En este proyecto se hace uso de ECS y del *Job System*.

2.6 ECS

ECS se basa en el principio de que los datos son más importantes que el código. En ECS, los objetos del juego se representan como entidades, que son colecciones de componentes. Los componentes son estructuras de datos simples que almacenan la información relevante para cada entidad. Los sistemas son funciones que operan sobre los componentes de las entidades y definen la lógica de la aplicación (9).

2.7 Job System

El *Job System* es una forma de escribir código que se ejecuta en paralelo en múltiples núcleos de procesador. Permite aprovechar el rendimiento de las CPU modernas sin tener que lidiar con los problemas de sincronización y seguridad de los hilos tradicionales. Se basa en el concepto de *jobs*, que son unidades de trabajo independientes que se pueden distribuir entre los núcleos disponibles. Los *jobs* se pueden crear y programar tanto para las entidades de ECS, como para los objetos convencionales de Unity (10).

3 Especificaciones y restricciones de diseño

La aplicación desarrollada en este proyecto se basa en un conjunto de sistemas que, mediante una arquitectura adecuada, conectan entre ellos de manera independiente. Dicha red de conexiones establece el algoritmo del programa.

El algoritmo consta de las siguientes etapas:

1. El usuario modifica un parámetro.
2. Todos los parámetros que dependen del parámetro modificado se actualizan de manera acorde (por ejemplo, si se modifica el vector de onda, la frecuencia angular se vería afectada).
3. Todas las modificaciones se aplican en la visualización en tiempo real.

En el desarrollo de la aplicación se tienen en cuenta las siguientes especificaciones:

- El programa tiene un Menú principal y acceso a dos tipos de simulación:
    - Simulación de ondas longitudinales acústicas.
    - Simulación de ondas longitudinales acústicas con tutorial previo.
- El usuario dispone de un tutorial de uso interactivo.
- El programa se puede ejecutar y utilizar en Windows, Mac OS y Linux.
- Durante la simulación, el usuario:
    - Dispone de una interfaz de usuario con la que cambiar los parámetros de la onda y visualizar dichos cambios en tiempo real.
    - Puede controlar la cámara a través del volumen de simulación.
- El programa tiene una arquitectura fácilmente escalable, para poder añadir nuevas funciones en el futuro.

La principal restricción de diseño que se ha planteado para el desarrollo de la aplicación ha sido la capacidad de procesado y renderizado de los dispositivos que vayan a reproducirla. Si se muestran demasiadas partículas en pantalla a la vez y el dispositivo no es potente, podría sufrir cuellos de botella de rendimiento, deteriorando enormemente la calidad de la visualización con ralentizaciones o congelaciones de pantalla o incluso podría colgarse completamente.

4 Descripción de la solución propuesta

A continuación, se describe el proceso de diseño, investigación y desarrollo de la solución propuesta.

Se han tomado determinadas decisiones de diseño a lo largo del proyecto con el fin de conseguir el mejor resultado posible para el objetivo marcado. Muchas de ellas han debido ser tomadas al inicio, puesto que determinan la configuración inicial del proyecto.

Se ha de elegir un motor gráfico de desarrollo para trabajar. Se han considerado como opciones viables los tres motores gráficos de desarrollo más populares, que son Unity, Unreal Engine y GODOT.

Unity es el más fácil de usar y tiene una gran comunidad de desarrolladores que ofrecen soporte y recursos. Sin embargo, Unity tiene algunas limitaciones en cuanto al rendimiento y la optimización, especialmente cuando se trata de escenas con muchos objetos y luces dinámicas, pero esto se compensa mediante el uso del sistema DOTS.

Unreal Engine es el más potente y avanzado de los tres, con un motor gráfico que ofrece una calidad visual superior al resto y un sistema de física que simula el comportamiento realista de las partículas. También tiene herramientas profesionales para el diseño y la programación, así como una licencia gratuita para proyectos no comerciales o con ingresos bajos. No obstante, Unreal Engine tiene una curva de aprendizaje muy alta y requiere un equipo con experiencia y conocimientos técnicos. Además, consume muchos recursos del sistema y puede ser difícil de adaptar a las necesidades específicas de cada proyecto.

GODOT es el más libre y flexible de los tres, con una licencia de código abierto que permite modificar y distribuir el motor como se desee. También tiene un sistema de nodos que facilita la creación y organización de escenas complejas, así como un lenguaje de programación propio que simplifica el desarrollo. Sin embargo, GODOT tiene una calidad gráfica inferior a la de los otros dos motores y un sistema de partículas menos sofisticado y eficiente. Además, tiene una comunidad más pequeña y menos recursos disponibles que los otros dos motores.

Teniendo en cuenta todo lo expuesto anteriormente, se ha seleccionado Unity como motor gráfico de desarrollo. Además, otro factor a favor de la utilización de Unity en este proyecto es la experiencia previa del alumno con este motor gráfico. Se ha empledado la versión LTS (*Long Term Support*) más reciente en el momento de comenzar con el desarrollo. En este caso es la versión 2021.3.6f1.

Otra de las decisiones de diseño es escoger un flujo de renderizado para trabajar. En la sección 2.3 se describen las diferentes opciones de flujo de renderizado. Dado que se requiere la máxima eficiencia y flexibilidad de rendimiento posible, se elige URP como flujo de renderizado.

A la hora de realizar el diseño se ha elegido que tenga compatibilidad con Windows, MacOS y Linux, ya que, entre esos tres sistemas operativos constatan la inmensa mayoría de usuarios de ordenador. Así se abarca un mayor número de posibilidades de compatibilidad para el usuario.

Para seleccionar el sistema de gestión de entradas de *Hardware* se han tenido en cuenta las diferentes opciones descritas en la sección 2.4 Por su mayor versatilidad y reescalabilidad se ha elegido el *Input System.*

Por último, se ha de elegir el IDE (*Integrated Development Environment)* a utilizar para la edición de los *scripts*. El IDE utilizado más comúnmente con Unity es Visual Studio, cuya versión de 2019 es la más optimizada para la versión de Unity elegida. Sin embargo, JetBrains Rider 2022 ofrece ciertas ventajas sobre Visual Studio:

- Tiene una interfaz más limpia y moderna, con menos distracciones y más opciones de personalización.
- Ofrece un rendimiento superior, especialmente en proyectos grandes, con una menor carga de memoria y CPU.
- Cuenta con herramientas avanzadas de refactorización, depuración, análisis de código y pruebas unitarias, que facilitan la escritura y el mantenimiento del código.
- Se integra perfectamente con Unity, permitiendo acceder a sus funciones y componentes desde el IDE, así como sincronizar los cambios en tiempo real.

La mayor desventaja que ofrece JetBrains Rider 2022 sobre Visual Studio es que requiere una licencia de pago.

Teniendo en cuenta todas las ventajas y desventajas planteadas se ha optado por JetBrains Rider 2022.

Con todo lo anterior se procede a la creación y configuración del proyecto para, por último, seleccionar la herramienta de control de versiones que se va a utilizar.

Una herramienta de control de versiones es un software que permite gestionar los cambios realizados en un conjunto de archivos o documentos a lo largo del tiempo. Con una herramienta de control de versiones, se puede mantener un historial de las modificaciones, revertir a versiones anteriores, resolver conflictos entre diferentes autores y colaborar de forma eficiente en un proyecto.

Git es una de las herramientas de control de versiones más populares y potentes que existen. Git tiene varias ventajas sobre otras herramientas, como, por ejemplo:

- Es un sistema distribuido, lo que significa que cada usuario tiene una copia completa del repositorio y puede trabajar de forma local sin depender de una conexión a internet o a un servidor central.
- Es rápido y eficiente, ya que utiliza un algoritmo de compresión que reduce el tamaño de los archivos y optimiza las operaciones de transferencia y sincronización.
- Es flexible y adaptable, ya que permite crear y gestionar diferentes ramas y etiquetas para organizar el trabajo según las necesidades de cada proyecto.
- Es seguro y confiable, ya que utiliza un sistema de identificación basado en claves criptográficas que garantiza la autenticidad e integridad de los datos.
- Es completamente compatible con el IDE seleccionado, pudiendo almacenar las versiones directamente desde ahí.

Teniendo en cuenta todas estas ventajas, se ha seleccionado Git como herramienta de control de versiones.

Para poder acceder al código fuente desde cualquier equipo, se ha de seleccionar una herramienta de almacenamiento en la nube de control de versiones.

Una herramienta de almacenamiento en la nube de control de versiones es un sistema que permite guardar, gestionar y compartir el código fuente de un proyecto de software.

GitHub es una de las plataformas más populares compatibles con Git y ofrece varias ventajas sobre otras opciones, tales como:

- Una interfaz web sencilla y amigable que facilita la navegación, la visualización y la edición de los archivos.
- Una gran comunidad de usuarios y desarrolladores que contribuyen con sus proyectos, comentarios, sugerencias y soluciones a problemas comunes.
- Una amplia variedad de funcionalidades integradas, como la gestión de ramas, etiquetas, solicitudes de extracción, revisiones de código, informes de errores, wikis, páginas web y acciones de GitHub.
- Una alta compatibilidad con otros servicios y herramientas, como JetBrains Rider, GitLab, Bitbucket, Visual Studio Code, Eclipse, Atom, etc.
- Una opción gratuita para proyectos públicos y de código abierto, así como planes de pago asequibles para proyectos privados y profesionales.

4.1 Teoría de ondas longitudinales planas

En este PFG se trabajará utilizando las hipótesis de la acústica lineal, que es una teoría sencilla para el sonido en fluidos, pero que puede describir la mayoría de los fenómenos acústicos comunes (1). Para ello se asumen las siguientes simplificaciones:

- Se supone que el fluido es homogéneo, isótropo y perfectamente elástico.
- No hay efectos disipativos, tales como los que surgen de la viscosidad o la conducción del calor.
- El análisis se limita a ondas de amplitud relativamente pequeña, de tal manera que los cambios de densidad serán también pequeños, comparados con su valor de equilibrio.
- La densidad de equilibrio constante en el fluido y la presión de equilibrio constante en el fluido tienen valores uniformes a través de este.

En este trabajo se va a simular más concretamente la propagación de ondas de sonido en el aire. Para ello, en la aplicación se mostrará el desplazamiento de las partículas en el medio, ya que es lo más intuitivo para los estudiantes, que son el objetivo de esta aplicación.

Existe una relación entre el desplazamiento de las partículas en el fluido y la variación de la presión en el mismo cuando se está propagando una onda acústica. Por esto, en la aplicación, el usuario introduce valores de presión acústica en la interfaz, y estos valores son convertidos internamente en valores de desplazamiento de las partículas.

La presión acústica o sobrepresión, p, se define como la diferencia entre la presión instantánea medida en un punto y la presión de equilibrio constante, que es la presión que se mediría si no se estuviese propagando una onda en ese medio.

La ecuación de onda lineal para la presión acústica se puede obtener a partir de otras tres ecuaciones, que son las siguientes:

1. Ecuación de estado:

$$p = B \cdot s \tag{1}$$

   Donde B es el módulo adiabático de volumen y s es la condensación en un punto o variación relativa de la densidad del medio cuando se propaga la onda acústica.

2. Ecuación de continuidad:

$$\nabla \cdot \vec{v}_p = -\frac{\partial s}{\partial t} \tag{2}$$

donde $\vec{v}_p$ es la velocidad de partícula, es decir, la velocidad con la que las partículas se desplazan cuando se está propagando una onda en el medio.

3. Ecuación de Euler Lineal:

$$\nabla p = -\rho_0 \frac{\partial \vec{v}_p}{\partial t} \tag{3}$$

donde ρ_0 es la densidad de equilibrio constante del fluido.

Combinando estas tres ecuaciones se obtiene una única ecuación diferencial que rige el comportamiento de la presión acústica en la propagación del sonido:

$$\nabla \cdot (\nabla \Phi) = \nabla^2 \Phi \rightarrow \nabla \cdot (\nabla p) = -\rho_0 \nabla \cdot \left(\frac{\partial \vec{v}_p}{\partial t}\right) = -\rho_0 \frac{\partial}{\partial t} \nabla \cdot \vec{v}_p = \rho_0 \frac{\partial^2 s}{\partial t^2} = \frac{\rho_0}{B} \frac{\partial^2 p}{\partial t^2} \tag{4}$$

Dando como resultado la siguiente ecuación diferencial para la presión acústica:

$$\nabla^2 p = \frac{1}{v_s^2} \frac{\partial^2 p}{\partial t^2} \tag{5}$$

Podemos comparar esta ecuación con la ecuación diferencial en derivadas parciales lineal de segundo orden que describe la propagación de una gran variedad de ondas:

$$\nabla^2 \Psi = \frac{1}{v_s^2} \frac{\partial^2 \Psi}{\partial t^2} \tag{6}$$

donde v_s es la velocidad de propagación de la onda en ese medio, yy cuya solución armónica es:

$$\Psi(\vec{r}, t) = \Psi_0 cos\,(\omega t - \vec{k} \cdot \vec{r} + \varphi) \tag{7}$$

donde Ψ_0 es la amplitud, ω es la frecuencia angular, $\vec{k}$ es el vector de onda, $\vec{r}$ es la posición, y φ es la constante de fase.

Por lo tanto, la solución armónica para la presión acústica es:

$$p(\vec{r}, t) = p_0 \cos\,(\omega t - \vec{k} \cdot \vec{r} + \varphi) \tag{8}$$

Y usando notación compleja se escribe como:

$$p(\vec{r}, t) = p_0 e^{i(\omega t - \vec{k} \cdot \vec{r} + \varphi)} \tag{9}$$

Esta solución describe una onda plana, ya que los frentes de onda son planos perpendiculares a la dirección de propagación.

Se puede deducir la función de onda para la velocidad de partícula a partir de la presión acústica utilizando la Ecuación de Euler Lineal, siendo el resultado:

$$\vec{v}_p = \frac{p_0}{\rho_0 v_s} e^{i(\omega t - \vec{k} \cdot \vec{r} + \varphi)} \vec{u} \tag{10}$$

donde $\vec{u}$ es la dirección de propagación.

Sabiendo que el desplazamiento de las partículas $\vec{\xi}$ y la velocidad de partícula se relacionan tal que:

$$\vec{v}_p = \frac{\partial \vec{\xi}}{\partial t} \tag{11}$$

Por lo tanto:

$$\vec{\xi} = \int \vec{v}_p dt \Rightarrow \vec{\xi} = \frac{p_0}{\rho_0 \omega v_s} e^{i(\omega t - \vec{k}\cdot\vec{r} + \varphi - \frac{\pi}{2})} \vec{u} \tag{12}$$

En notación armónica es:

$$\vec{\xi} = \frac{p_0}{\rho_0 \omega v_s} \cos(\omega t - \vec{k}\cdot\vec{r} + \varphi - \frac{\pi}{2})\vec{u} \tag{13}$$

De lo cual se deduce que el desplazamiento está retrasado $\frac{\pi}{2}$ con respecto a la presión y que la relación entre la amplitud del desplazamiento de las partículas y la de la presión acústica es:

$$\xi_0 = \frac{p_0}{\rho_0 \omega v_s} \tag{14}$$

Esta es la relación que se utilizará en la aplicación para realizar la conversión de presión acústica (dato que introduce el usuario) a desplazamiento de las partículas (que es lo que se visualiza en la aplicación).

Puesto que se ha seleccionado el aire como medio en el que se propagarán las ondas, se utilizarán los siguientes valores para la densidad de equilibrio constante del fluido y para la velocidad del sonido (11):

- ρ_0 = 1.2041 $^{kg}/_{m^3}$
- v_s = 343 $^{m}/_{s}$

4.2 Movimiento de partícula como GameObject

4.2.1 La clase MonoBehaviour

Antes de comenzar a desarrollar el comportamiento de las partículas, se ha de entender la clase principal que utiliza Unity para manejar lógica en tiempo real, que es *MonoBehaviour*.

La clase *MonoBehaviour* es una clase especial que hereda de la clase *Behaviour* de Unity. Esta clase permite crear scripts que se pueden adjuntar como componentes a los *GameObjects* y acceder a sus atributos y eventos. La clase *MonoBehaviour* dispone de varias funciones que se ejecutan en diferentes momentos del ciclo de vida de un *script*, como *Awake*, *Start*, *Update*, *FixedUpdate*, *LateUpdate*, *OnEnable*, *OnDisable*, *OnDestroy*, etc. Estas funciones permiten inicializar variables, actualizar el estado del script, manejar la física, la entrada del usuario, la interfaz gráfica y mucho más (12).

4.2.2 Los GameObjects en Unity

Para poder entender mejor esta sección se ha de definir qué es un *GameObject* en Unity.

Un *GameObject* es un objeto que puede representar cualquier elemento en una escena de Unity, como personajes, luces, cámaras, efectos especiales, etc. Un *GameObject* por sí solo no es visible y, aunque se puede usar como punto de referencia en la escena para varios usos, no tiene una funcionalidad específica. La funcionalidad de un *GameObject* es definida por los componentes que se le agreguen a éste. Los componentes son clases con propiedades y comportamientos específicos. Por ejemplo, para crear una luz en la escena,

se debe agregar un componente de luz a un *GameObject* vacío. Unity cuenta con muchos componentes integrados como renderizadores, colisionadores, audio, física, etc. También se pueden crear componentes personalizados usando la API de scripting de Unity (13).

4.2.3 Tipos de movimiento

Existen múltiples matices a la hora de mover un elemento en Unity, pero todos esos matices se pueden clasificar en dos ramas principales:

- Movimiento por *Transform*
- Movimiento por física

Antes de continuar hay que aclarar que la *Transform* de un objeto es un componente que recoge la información de posición, rotación y escala de dicho objeto en las 3 dimensiones de Unity.

El movimiento por *Transform* engloba todos aquellos métodos que basan su funcionamiento en cambiar las coordenadas de posición y/o rotación de la *Transform* de un objeto. Los métodos más destacados son por programación o mediante animaciones que modifican la posición del hueso raíz.

El movimiento por física engloba los métodos que basan su funcionamiento en la aplicación de diferentes fuerzas sobre el objeto a mover. Hace uso de un componente de cuerpo rígido, que aporta propiedades físicas (se ve afectado por las fuerzas del entorno) a un objeto. Los dos métodos principales son el de cuerpo rígido normal (el entorno afecta al movimiento) o el cuerpo rígido cinético (el entorno no afecta al movimiento).

4.2.4 Moviendo la partícula

En este caso particular se va a utilizar movimiento por *Transform*, modificando la posición de cada partícula de manera independiente a partir de la ecuación de onda plana longitudinal.

Para ello se crea una clase *Particle*, que hereda de Monobehaviour, para poder aplicarla como componente en un GameObject, con los atributos de la ecuación: amplitud, frecuencia angular y fase inicial de tipo *float* y el vector de onda $\vec{k}$, la posición inicial en el mundo $\vec{r}$ y el vector de onda $\vec{k}$ normalizado de tipo *Vector3* (*struct* del espacio de nombres *UnityEngine* de la API de Unity) (14). Para el tiempo t se usa el atributo *time* de la clase *Time* del espacio de nombres *UnityEngine*, que devuelve el valor en segundos que han pasado desde el inicio de la ejecución de la aplicación.

Para conocer la posición inicial en el mundo $\vec{r}$ se utiliza la función *Start* de la clase *Monobehaviour*, que se ejecuta justo antes del primer *frame* en el que el objeto está activo. En *Start* se almacena el atributo *position* de la clase *Transform* en $\vec{r}$, siendo esta la posición inicial.

Para actualizar la posición de la partícula en tiempo real, se utiliza la función *Update* de la clase *Monobehaviour*, la cual funciona como un bucle infinito que se ejecuta una vez cada *frame*. La ecuación de onda devuelve en cada *frame* un valor instantáneo entre amplitud y -amplitud en referencia a la posición de equilibrio $\vec{r}$. Dicho valor instantáneo, al ser una onda longitudinal, tiene la misma dirección que el vector $\vec{k}$ normalizado. El cálculo de la nueva posición sigue los siguientes pasos:

1. Cálculo del valor instantáneo de la partícula

$$valorInstantaneo = amplitud * \cos(frecuenciaAngular * Time.time - (kx * rx + ky * ry + kz * rz) + faseInicial)$$

2. Conversión del valor instantáneo a vector en la dirección de k normalizado

$$vectorValorInstantaneo = valorInstantaneo * kNormalizado$$

3. Aplicación del valor instantáneo con respecto a la posición inicial

$$Transform.position = vectorValoInstantáneo + r$$

Con esta lógica, cualquier *GameObject* que tenga como componente la clase *Particle* ya realiza un movimiento armónico simple en la dirección de $\vec{k}$.

4.3 La apariencia de la partícula

Antes de desarrollar la toma de decisiones con respecto a la apariencia de la partícula, se ha de entender el flujo de renderizado y cómo se dibuja en pantalla aquello que está en el espacio 2D de la cámara. Primero se van a definir las partes que van a entrar en juego en este proceso: la malla 3D, el material y el *shader*. Normalmente para este proceso entrarían en juego todos los elementos que tienen que ver con la iluminación, como la luz y las sombras, pero para ahorrar recursos se ha optado por un renderizado sin iluminación.

4.3.1 Malla 3D

Una malla es una representación gráfica de un objeto tridimensional compuesta por vértices, aristas y caras. Los vértices son los puntos que definen la forma del objeto, las aristas son las líneas que conectan los vértices y las caras son las superficies planas que forman el volumen del objeto. Una malla puede tener diferentes niveles de detalle y complejidad según el número y la distribución de sus elementos. Las mallas se utilizan en el mundo del 3D para modelar, animar y renderizar objetos virtuales (15).

4.3.2 Shader

Un *shader* es un programa que determina cómo se renderiza el modelo en la pantalla, y una serie de propiedades que se pueden ajustar para modificar el efecto de este. Para crear y modificar *shaders* en Unity se puede usar el lenguaje HLSL (High Level Shader Language) o el paquete *Shader Graph* de Unity, que permite hacerlo de manera gráfica (16).

4.3.3 Material

Un material en Unity es un objeto que define el aspecto visual de un modelo 3D. Un material contiene una referencia a un *shader*. Por ejemplo, un material puede tener una propiedad de color, una propiedad de textura, una propiedad de metalicidad, una propiedad de suavidad, etc. Estas propiedades se pueden cambiar en el editor de Unity o mediante código en tiempo de ejecución. Un material se puede asignar a uno o más objetos 3D para darles el mismo aspecto visual (17).

4.3.4 Definiendo la apariencia

Si se piensa en dar apariencia a una partícula, probablemente lo primero que viene a la mente es una esfera. Sin embargo, la malla de una esfera tiene 520 caras que, para el objetivo de ahorro de rendimiento, no es nada conveniente, ya que, si se plantea una rejilla

de 16x16x16 partículas, habría que renderizar 2.129.920 caras. Esto en un videojuego convencional lleno de elementos estáticos es un número más que razonable, pero en este caso todos los elementos van a estar moviéndose y ejecutando lógica.

Se podría plantear reducir el número de caras de la esfera, haciendo que ésta tenga menos resolución, pero existe un método mucho más eficiente y usado constantemente en la industria de los videojuegos, que es el *Billboard*.

El *Billboard* consiste en hacer que los vértices de un objeto miren constantemente a la cámara, de manera que la normal de una imagen plana siempre tendrá la dirección contraria a la que mira la cámara. Esto se consigue con un *shader* en el que se multiplican escalarmente las componentes x, y, z de la dirección a la que mira la cámara y la posición de cada vértice respectivamente y recomponiéndolos en un solo vector mediante una suma, resultando en la proyección de cada vértice hacia la cámara (figura 1).

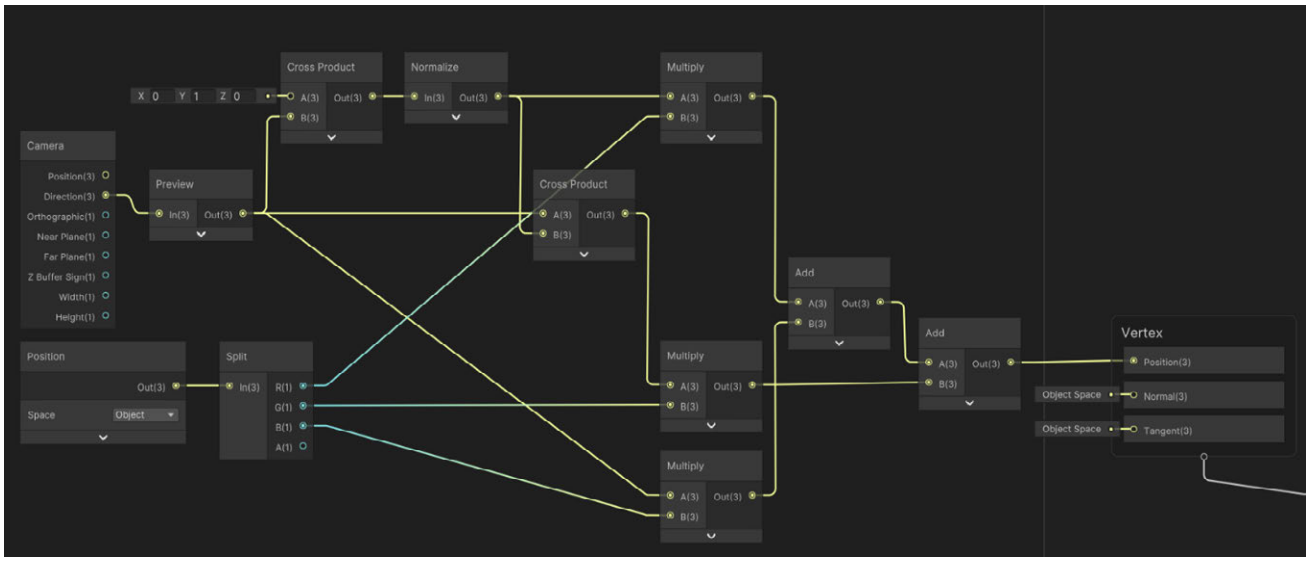

Figura 1: Configuración matemática de un *shader Billboard*

Siendo la imagen insertada un círculo con diferentes niveles en el canal *Alpha*, se puede simular una esfera con una malla de una cara, en vez de 520 (figura 2), reduciendo masivamente el rendimiento requerido para renderizar grandes cantidades de este objeto.

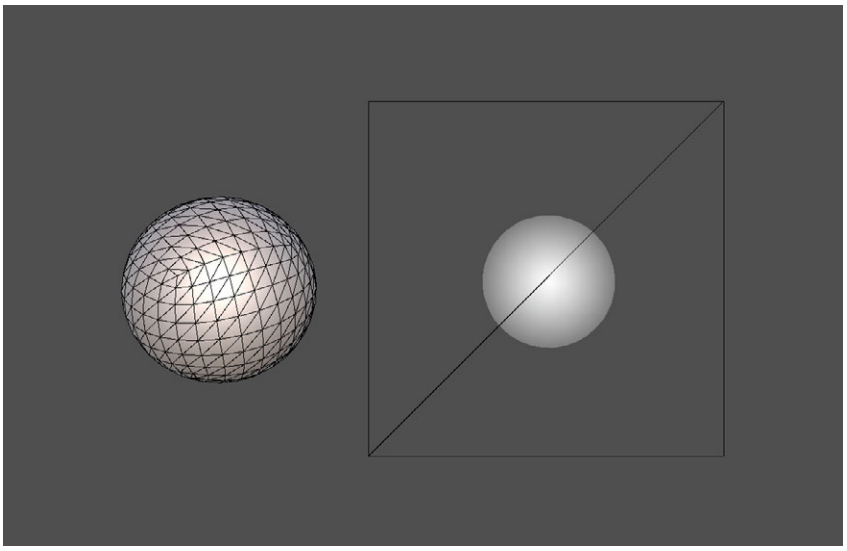

Figura 2: Comparación gráfica entre esfera y *quad*

4.4 El manejador de partículas

Dado que en la aplicación final se han de poder modificar los parámetros de la onda en tiempo real, se crea un manejador para dichos parámetros al que se conectan los atributos de cada partícula. De esta manera, al modificar un parámetro en el manejador, se modifica en cada partícula individual al mismo tiempo.

Para ello se crea la clase *ParticleManager* que hereda de *MonoBehaviour*. Dicha clase, al ser un manejador, tiene un atributo estático de su propia clase actúa como instancia única en la aplicación. Puesto que *ParticleManager* hereda de *MonoBehaviour*, puede actuar como componente en cualquier objeto de la escena por lo que, por accidente, podría haber más de una instancia. Para asegurar que dicha instancia es única se hace uso de la función *Awake* de *MonoBehaviour*, la cual se ejecuta cuando se activa el objeto por primera vez y antes que *Start*. En dicha función se usa la siguiente lógica:

```
Si Instance es null
     Esta instancia es Instance
Si no
     Destruir este objeto
```

De esta manera, si se activa un objeto con este *script* entre sus componentes y ya había uno en la escena, el nuevo objeto se destruirá, asegurando que no haya más de un manejador.

El manejador de partículas tiene los mismos atributos que las partículas, con algunos añadidos.

Se añaden tres constantes de tipo *float*:

Un modificador de 0,001 para la frecuencia angular ya que, si se intentara ver una partícula moverse a 1000π radianes por segundo (500Hz) en un monitor de ordenador, no se podría, puesto que los más potentes en el mercado llegan a una tasa de refresco de, como máximo, 360Hz.

Un modificador de 100000 para el desplazamiento. Sin este modificador el desplazamiento sería imperceptible por sus magnitudes diminutas.

Densidad del aire de 1.2041. Se usa para calcular el desplazamiento a partir de la presión.

Se añaden dos variables de tipo *float*:

El desplazamiento y la velocidad del sonido.

Se añade un atributo de clase *Action*:

OndaActualizada, el cual es invocado cada vez que un atributo es modificado para que el resto de *los scripts* sean informados y actúen en consecuencia.

Puesto que algunos de los atributos dependen de otros, se crean diversos métodos de actualización de atributos para que calculen los cambios que han de hacerse tras modificar un parámetro en el resto de ellos.

4.5 Adaptación de partículas de GameObjects a entidades

Una vez definida la lógica básica del movimiento armónico de las partículas y con el fin de mejorar el rendimiento de la aplicación, hay que adaptar la lógica desarrollada en la sección 4.2 a la arquitectura ECS.

Como se ha explicado en la sección 2.6, la arquitectura ECS se divide en tres partes: Entidades, componentes y sistemas.

4.5.1 Entidades

Para generar las entidades se requieren varios elementos básicos. Las entidades no se crean en la escena de Unity en sí, si no en un mundo especial que se inyecta en la escena de Unity para ser renderizado. Dicho mundo cuenta con un manejador de entidades que actúa sobre todas las entidades generadas en dicho mundo. De esta forma y, aunque en este proyecto sólo se requiera un mundo, se pueden tener múltiples mundos en paralelo manejando múltiples tipos de entidades.

Para llevar esto a cabo se crea la clase *ParticleSpawner*, que hereda de *MonoBehaviour*. La clase *ParticleSpawner* tiene un comportamiento similar al de un manejador, por lo que se realiza la misma lógica que en la clase *ParticleManager* para que tenga una única instancia estática, añadiendo un atributo estático de su propia clase con el nombre *Instance*.

Puesto que se pretende generar una rejilla de partículas en el volumen de la escena, son necesarios atributos de tipo *float* para los tamaños mínimos y máximos de la rejilla, los tamaños por separado para los ejes x, y, z (cantidad de partículas por eje) y el espaciado en unidades de Unity entre las partículas.

Para generar las entidades se necesitan tres elementos del espacio de nombres *Unity.Entities*:

Un objeto del *struct Entity*, que alberga la conversión a entidad de un *GameObject*, en este caso, una partícula.

Un objeto del *struct EntityMaganger*, que guarda la instancia estática del manejador de entidades.

Un objeto de la clase *World*, en el que se instancia el mundo en el que se generan las entidades. Cada objeto de la clase *World* tiene como parámetro una instancia estática de *EntityManager* (9).

Para los atributos del generador de partículas se requiere un *GameObject* con las características requeridas para convertirse a entidad. Dicho objeto se describe más adelante.

Se crea también un atributo de la clase *Action*, el cual es invocado cada vez que se reinicia la rejilla para que el resto de *los scripts* sean informados y actúen en consecuencia.

En la función *Start* de *Monobehaviour* se inicializa el mundo. A partir de dicho mundo se inicializa en manejador de entidades y, por último, se convierte el *GameObject* prefabricado a entidad.

Se crean métodos para instanciar una entidad en una posición determinada y para instanciar una rejilla de entidades en función de los atributos de tamaño de rejilla en cada eje y el espaciado entre cada entidad, haciendo uso del método anterior, cuya lógica es:

```
para i = 0, i < tamaño en X, i++)
    para (j = 0, j < tamaño en Y, j++)
        para (k = 0, k < tamaño en Z, k++)
            Isntanciar entidad (espaciado * (i, j, k))
```

De manera que si, por ejemplo, se instanciara una rejilla de 16x16x16 con un espaciado de 0,5, la primera entidad estaría en la posición (0, 0, 0) y la última en la (8, 8, 8).

También se crean otros dos métodos:

Uno para destruir la rejilla y otro para reiniciarla, que hace uso del primero y después vuelve la vuelve a instanciar.

Por último, se crean métodos para poder modificar los parámetros de la rejilla desde clases externas. Cada vez que se cambia un atributo la rejilla se reinicia.

4.5.2 Componentes

Para que una entidad pueda tener características, se ha de agregarle componentes. Dichos componentes se pueden adjuntar a un *GameObject* prefabricado.

Para que la entidad se pueda renderizar necesita los componentes de Unity *Mesh Filter* y *Mesh Renderer.*

Mesh Filter define la malla que se va a renderizar y *Mesh Renderer* cómo se va a renderizar (es donde se agrega el material con su *shader* correspondiente).

Por último, para añadirle atributos a la entidad se crea una clase *AcousticWaveData* que hereda de *IComponentData*. La interfaz *IComponentData* permite al sistema de entidades, que, como se ha explicado en la sección 2.6, es el que define la lógica a la entidad, poder operar con los atributos de cada entidad por separado.

Se añaden los mismos atributos que en la clase *Particle*, se conectan con los valores del manejador de partículas, con la diferencia de que los atributos de tipo *Vector3*, ahora son de tipo *float3*, del espacio de nombres *Unity.Mathematics*, que es el tipo con el que opera la arquitectura ECS (18).

4.5.3 Sistema

Para definir la lógica de las entidades se ha de crear un sistema. Todos los sistemas son clases parciales que heredan de *SystemBase*

Una clase parcial es una característica de C# que permite definir una clase en varios archivos o secciones. Debe tener el mismo nombre, espacio de nombres y accesibilidad en todos los archivos donde se define (en este caso sólo hay una, pero si hubiese más, todas heredarían de *SystemBase*). El compilador combina todas las partes de una clase parcial en una sola clase en el momento de compilar el código.

SystemBase es una clase abstracta del espacio de nombres *Unity.Entities*. Representa un sistema de componentes (9).

Un sistema de componentes es una unidad de código que se ejecuta en cada entidad que tiene los componentes requeridos por el sistema. La clase *SystemBase* proporciona métodos y propiedades para acceder a los datos de las entidades, gestionar las dependencias entre sistemas, y controlar el ciclo de vida del sistema.

En este caso se usan los métodos *OnStartRunning* y *OnUpdate*, que funcionan igual que los métodos *Start* y *Update* de *MonoBehaviour*.

Se crea la clase parcial *AcousticWaveSystem*, que hereda de *SystemBase*.

Puesto que, hasta que no se ha generado cada entidad, el atributo posición inicial en el mundo $\vec{r}$ aún no contiene información, se crea un método para establecer la posición inicial de cada entidad.

En el método *OnStartRunning* se ejecuta el método anterior y se realiza una suscripción a la acción de *ParticleSpawner* que se invoca cada vez que se reinicia la rejilla, para que cada vez que se invoque se establezcan las posiciones iniciales de la rejilla nueva.

En el método *OnUpdate* se actualiza el atributo de tiempo y se aplica la misma lógica de movimiento que en *Particle*, explicada en la sección 4.2. La diferencia con la clase *Particle* es que los sistemas no trabajan con la clase *Time* del espacio de nombres, si no con el atributo *Time* de tipo *TimeData* de la propia clase *SystemBase*. Por lo que ahora el tiempo está definido por *Time.ElapsedTime*, que funciona exactamente igual que *Time.time*.

Cabe destacar la diferencia de arquitecturas de la lógica establecida inicialmente con *GameObjects* y la descrita en esta sección:

En la lógica inicial hay una cantidad de partículas determinada, cada una con una instancia de la clase *Particle* y cada una ejecutando su lógica individualmente en cada fotograma. Con esta nueva arquitectura, *AcousticWaveSystem* gestiona la lógica de todas las partículas a la vez, repartiendo el peso de rendimiento de manera paralela entre los núcleos del microprocesador del equipo en el que se ejecuta.

4.6 Configuración del InputSystem

Se ha diseñado un movimiento de cámara por el volumen de simulación inspirado en los de Unity y Blender.

Se crea un *Asset* de tipo *InputActions* y se procede a su configuración.

Se configura un mapa de acciones de jugador con las siguientes características:

Para permitir el movimiento y rotación de la se usa el botón derecho del ratón.

Para mover la cámara en los ejes x y z (horizontalmente) se usan las teclas W, A, S y D o las flechas del teclado.

Para mover la cámara en el eje y (verticalmente) se usan las teclas Q y E.

Para rotar la cámara se usa la delta de movimiento del ratón.

Para poder panear la cámara se usa el botón central del ratón.

Para el paneo de la cámara se usa la delta de movimiento ratón.

Se añade una tecla de reinicio de posición de la cámara para el que se usa la F.

Se usa la tecla *Escape* para abrir y cerrar el menú de pausa.

Se configura un mapa de acciones para la interfaz de usuario con las acciones por defecto de Unity.

Los mapas de acciones se activan y desactivan cuando se necesitan, de manera que, por ejemplo, cuando el usuario esté en un menú el mapa de acciones de jugador esté desactivado y no pueda mover la cámara por accidente en ese momento.

Las entradas de *Hardware* mediante el *InputSystem* se pueden manejar de varias maneras:

1. Suscripción a cualquiera de los tres eventos de cada acción
    a. Iniciada: Se invoca cuando se pulsa la tecla asociada a la acción
    b. Realizada: Se invoca cuando se cumplen los requisitos asociados a la acción (por ejemplo, mantener pulsada una tecla). Si no hay requisitos, se invoca a la vez que el evento “iniciada”.
    c. Cancelada: Se invoca cuando se suelta la tecla asociada a la acción.
2. Lectura del valor instantáneo de cada acción

Se genera una clase *Default_Input_Actions* a partir del *Asset* configurado para poder manejar las acciones del *InputSystem* desde código, en vez de desde la interfaz de Unity.

Se crea un *GameObject* con el componente de tipo *Player Input*. Dicho componente permite a un objeto de la escena manejar los eventos de las acciones creadas en el *Asset* de tipo *InputActions*.

4.7 El Manejador de entradas

Para manejar las acciones y valores de entrada del *InputSystem* se crea una clase *InputManager* que hereda de *MonoBehaviour*, que se coloca como componente en el mismo *GameObject* que el componente *Player Input* y que actúa de intermediaria entre la clase generada por el *Asset* de tipo *InputActions* y el resto del programa.

La clase *InputManager* es un manejador, por lo que se realiza la misma lógica que en la clase *ParticleManager* para que tenga una única instancia estática, añadiendo un atributo estático de su propia clase con el nombre *Instance*.

Se añade un atributo de la clase *Default_Input_Actions* para poder acceder a los eventos y valores de entrada.

Se añaden los atributos de la clase *Action* necesarios para gestionar los eventos de entrada que se van a usar. En este caso son:

- Permitir movimiento realizada y cancelada (pulsar y soltar el botón derecho del ratón)
- Permitir paneo realizada y cancelada (pulsar y soltar la rueda del ratón)
- Reinicio de cámara realizada (pulsar la tecla F)
- Pausa realizada (Pulsar la tecla *Escape* en la simulación)
- Reanudar realizada (Pulsar la tecla *Escape* en el menú)

En el método *Awake* de *MonoBehaviour*, además de la lógica de la instancia estática, se instancia el atributo de la clase *Default_Input_Actions* y se activa el mapa de acciones del jugador.

En el método *Start* de *MonoBehaviour* se realizan las suscripciones a las acciones mencionadas anteriormente procedentes del atributo de tipo *Default_Input_Actions* para que, cuando sean invocadas, invocar las creadas en la propia clase *InputManager*.

Se crean métodos públicos para leer el valor de las acciones de movimiento horizontal y vertical, así como el valor de la delta de movimiento del ratón.

Se crean también métodos públicos para cambiar del mapa de acciones de jugador al de interfaz de usuario y viceversa.

Con esto ya está todo listo para poder interpretar las entradas de *Hardware* desde cualquier punto del programa.

4.8 El controlador de la cámara

La cámara en una escena de Unity es un *GameObject* que contiene el componente *Camera*. El componente Camera de Unity es el que permite visualizar la escena desde el punto de vista de un objeto. Tiene varias propiedades que se pueden configurar para ajustar el aspecto de la imagen, como el campo de visión, la distancia de recorte, el color de fondo, etc.

4.8.1 Jerarquía de una escena en Unity

Antes de continuar se ha de entender cómo funciona la jerarquía de una escena en Unity.

La jerarquía en una escena de Unity es la forma de organizar los objetos que componen la escena. Cada objeto, como ya se ha explicado, tiene un nombre y una serie de componentes que definen su comportamiento y apariencia. La jerarquía permite crear relaciones entre los objetos, como la herencia y el anidamiento. La herencia significa que un objeto puede heredar las propiedades y los componentes de otro objeto que sea su padre. El anidamiento significa que un objeto puede contener otros objetos como hijos, formando una estructura jerárquica. Si se mueve el padre, se mueven los hijos y si el padre rota, los hijos rotan a su alrededor.

El *GameObject* de la cámara se coloca como hijo de otro *GameObject*, que va a ser el que contenga el script con la lógica de movimiento y rotación.

4.8.2 Creando el script

Para controlar la cámara existen múltiples opciones. En este caso se ha elegido un control de pájaro (poder volar libremente por el volumen de la escena) en primera persona con posibilidad de paneo inspirado en el control de cámara de Unity y Blender.

Para llevarlo a cabo se crea una clase *CameraSystem* que hereda de *MonoBehaviour*.

Se crean atributos de tipo *float* para las velocidades de movimiento, rotación y paneo.

Se crea un atributo de tipo *Vector3* para la posición de reinicio de la cámara.

Se crea un atributo de tipo *Quaternion* (*struct* del espacio de nombres *UnityEngine* de la API de Unity) para la rotación de reinicio de la cámara.

Se crean dos atributos de tipo *bool* que determinan si la cámara tiene permitido moverse y/o panear.

Se crean dos atributos de tipo *bool* para activar y desactivar la inversión de los ejes de rotación de la cámara en x e y (la cámara no rota alrededor del eje z, porque si no se inclinaría, haciendo muy incómoda la perspectiva)

Se crean dos atributos de tipo *int* como factor de inversión de cada eje. Se conectan a los booleanos mediante un operador condicional. Si sus booleanos correspondientes son *true*, valen 1. Si son *false*, valen -1.

En la función *Start* de *Monobehaviour* se toman las referencias de reinicio de posición y rotación de la cámara con su posición y rotación iniciales. También se realizan las suscripciones a los eventos de permitir movimiento realizada y cancelada, permitir paneo

realizada y cancelada y reinicio de cámara realizada de la instancia estática de *InputManager*.

Para los movimientos de cámara se utiliza la función *LateUpdate* de *Monobehaviour* que funciona igual que *Update*, pero es lo último que se ejecuta al final de cada *frame*. De esta manera la cámara siempre será la última en moverse, tras renderizar el resto de los objetos, evitando así que haya efectos visuales extraños por desincronización de movimientos.

En *LateUpdate* se ejecuta la siguiente lógica:

```
Si panearPermitido
        PanearCamara
Si moverPermitido
        MoverCamara
        RotarCamara
```

4.8.3 La lógica de paneo

Para lógica de paneo de la cámara se ha de convertir el vector de dos dimensiones normalizado que devuelve la delta de movimiento del ratón en un vector en el plano de xy local de la cámara.

Para ello se realizan los siguientes pasos:

1. Lectura del valor instantáneo de la delta de movimiento del ratón

$$valorPanVector = InputManager.Instance.LeerDeltaRaton$$

2. Conversión del vector del ratón al plano local xy de la cámara

$$direccionPan = -xCamara * valorPanVector.x - yCamara * valorPanVector.y$$

3. Aplicación del movimiento:

$$Transform.Mover(direccionPan * velocidadPan * Time.DeltaTime)$$

Time.DeltaTime es el tiempo que ha transcurre entre cada *frame*.

Puesto que cada monitor y equipo puede tener diferentes frecuencias de refresco de fotogramas, se usa *Time.DeltaTime* para normalizar el valor de cualquier elemento dentro de *Update* y *LateUpdate*.

4.8.4 La lógica de movimiento

La lógica de movimiento de la cámara es parecida a la de paneo, pero entran más factores en juego. Se han de convertir el vector de dos dimensiones normalizado que devuelve el movimiento horizontal y el *float* limitado entre -1 y 1 que devuelve el movimiento vertical en un vector de tres dimensiones, el cual se ha de aplicar a los ejes locales de la cámara para determinar el movimiento final de la misma.

Para ello se realizan los siguientes pasos:

1. Lectura del valor instantáneo del movimiento horizontal

$$moverHorizontalVector = InputManager.Instance.LeerMovimientoHorizontal$$

2. Lectura del valor instantáneo del movimiento vertical

$$moverVerticalfloat = InputManager.Instance.LeerMovimientoVertical$$

3. Conversión de los valores de entrada de movimiento a un vector de tres dimensiones

$$moverTotalVector = nuevo\ Vector3(moverHorizontalVector.x, moverVerticalFloat, moverHorizontal.y)$$

4. Conversión a ejes locales de la cámara:

$$direccionMovimiento = xCamara * moverTotalVector.x + yCamara * moverTotalVector.y + zCamara * moverTotalVector.z$$

5. Aplicación del movimiento:

$$Transform.Mover(direccionMovimiento * velocidadMovimiento * Time.DeltaTime)$$

4.8.5 La lógica de la rotación

Para lógica de rotación de la cámara se ha de convertir el vector de dos dimensiones normalizado que devuelve la delta de movimiento del ratón en un vector de rotación en el plano xy global, teniendo en cuenta los factores de inversión de ejes.

Para ello se realizan los siguientes pasos:

1. Lectura del valor instantáneo de la delta de movimiento del ratón

$$valorRotacionVector = InputManager.Instance.LeerDeltaRaton$$

2. Conversión del vector del ratón al plano local xy de la cámara

$$direccionRotacion = xGlobal * (valorRotacionVector.x * xfactorInversion) + yGlobal * (valorRotacionVector.y * yfactorInversion)$$

3. Se asegura que la componente z de la dirección de rotación es cero

$$direccionRotacion.z = 0$$

4. Aplicación de la rotación:

$$Transform.Rotar(direccionRotacion * velocidadRotacion * Time.DeltaTime)$$

4.8.6 Lógica de las acciones de entrada

En la función *Start* de *Monobehaviour* se han hecho suscripciones a acciones de la instancia estática de *InputManager* para las que hay que actuar en consecuencia. Se crean los siguientes métodos para cada acción:

- Cuando se activa permitir movimiento realizada, el *bool* moverPermitido es *true*
- Cuando se activa permitir movimiento cancelada, el *bool* moverPermitido es *false*
- Cuando se activa permitir paneo realizada, el *bool* panearPermitido es *true*
- Cuando se activa permitir paneo cancelada, el *bool* panearPermitido es *false*
- Cuando se activa reinicio de cámara realizada, se devuelve la cámara a su posición y rotación iniciales

La cámara ya puede moverse libremente por el volumen de simulación usando las entradas de *Hardware* de teclado y ratón.

4.9 La interfaz de usuario

La UI (*User Interface*) en Unity es el conjunto de elementos gráficos que permiten al usuario interactuar con el juego o la aplicación. Para crear una UI se necesita un *Canvas*, que es un componente que define el espacio donde se pueden colocar los elementos de la UI y que es renderizado de manera independiente a la escena. El *Canvas* se puede configurar de diferentes maneras según el tipo de cámara y la resolución que se quiera usar. Dentro del *Canvas* se pueden añadir elementos como botones, imágenes, textos, etc. Cada elemento tiene un componente llamado *Rect Transform*, que define su posición, tamaño, rotación y anclaje dentro del *Canvas*. El *Rect Transform* permite ajustar los elementos de la UI según el tamaño y la orientación de la pantalla, así como crear diseños responsivos y dinámicos. Puede haber multitud de *Canvas* en una escena.

Al componente *Canvas* siempre lo acompañan dos componentes:

- El componente *Canvas Scaler*, que permite que el *Canvas* pueda escalar sus elementos en función de la resolución de la pantalla en la que se reproduce la aplicación
- El componente *Graphic Raycaster*, que permite interactuar con los elementos del *Canvas* con los *clicks* del ratón.

En todos los *Canvas* que se crean en este proyecto, el *Canvas Scaler* está configurado para escalar sus elementos en función de la resolución de pantalla, con una referencia original de 1920x1080.

4.9.1 El Canvas de simulación

Se crea un *Canvas* destinado a la UI que aparece durante la simulación.

Se crea un panel de configuración de la onda. Un panel es un elemento de la interfaz de usuario que se utiliza para organizar y agrupar otros elementos UI dentro de un *Canvas*. En este panel se colocan los campos de entrada de los parámetros de la onda.

Se crean dos objetos prefabricados para los campos de entrada:

- Campo de entrada *Vector3*
- Campo de entrada *float*

Para el campo de entrada *Vector3* se crea un panel de 375x150. Como hijos de ese panel se crean cuatro textos. Uno para el nombre del parámetro y los otros tres para sus componentes. También se crean como hijos del panel tres entradas de texto. En la figura 3 se puede ver la distribución de los elementos dentro del panel.

Figura 3: Distribución del campo de entrada *Vector3*

Para el campo de entrada *float* se crea un panel de 500x150. Se añade un componente de alineación horizontal en el panel, que sirve, como su propio nombre indica, para alinear de forma automática los elementos hijos del panel. Como hijos de ese panel se crean dos textos. Uno para el nombre del parámetro y los otro para su símbolo. También se crea como hijo del panel una entrada de texto. En la figura 4 se puede ver la distribución de los elementos dentro del panel.

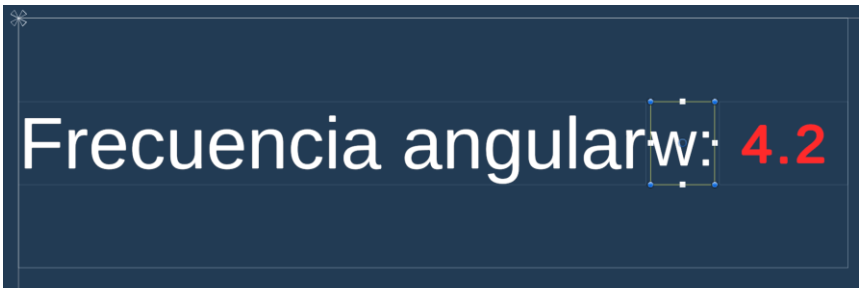

Figura 4: Distribución del campo de entrada *float*

Una vez creados los objetos prefabricados de los campos de entrada, se ponen en el panel de configuración de la onda y se distribuyen en la pantalla como se ilustra en la figura 5.

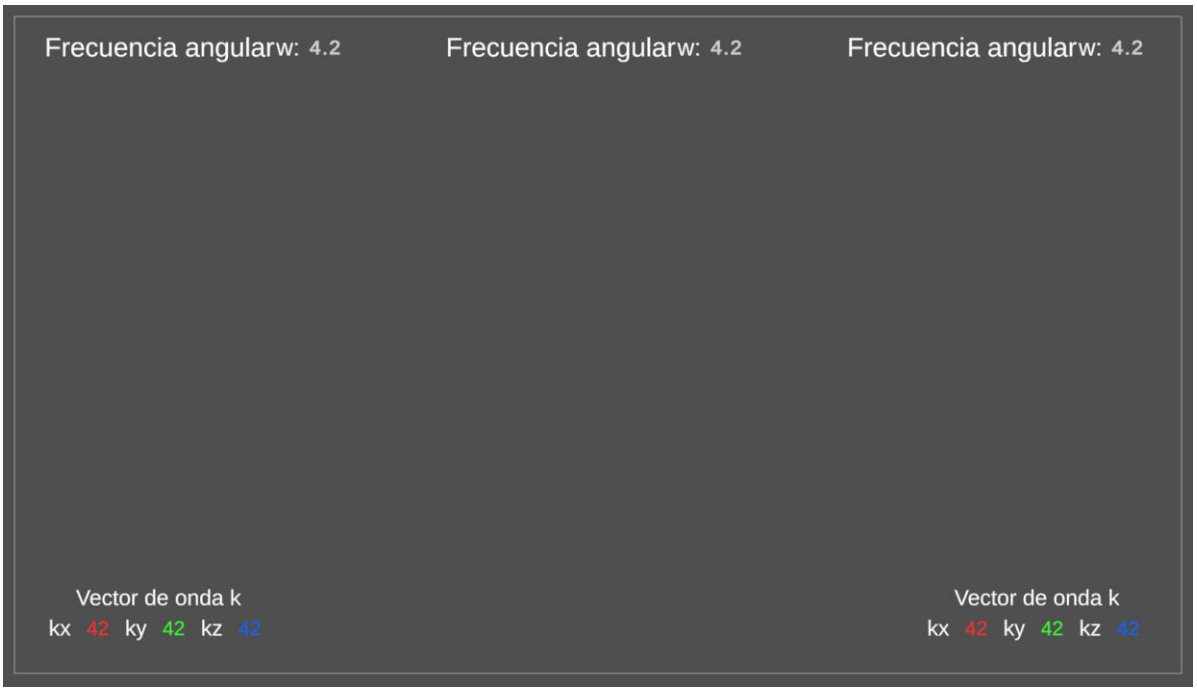

Figura 5: Distribución de elementos en el panel de configuración

Se crea un botón de configuración. Un botón en la UI de Unity es un componente que permite al usuario iniciar o confirmar una acción al hacer clic sobre él. Un botón tiene una imagen, un texto y un evento que se activa cuando el usuario lo presiona y/o lo suelta. El componente *Button* tiene varias propiedades que determinan su apariencia y su comportamiento, como *Interactable*, *Transition*, *Navigation* y *OnClick*. En este caso particular se sustituye el texto por una imagen en formato PNG con el fondo transparente de un engranaje. De manera que la imagen blanca de fondo del botón cambia de color cuando se interactúa con él y la imagen del engranaje es fija.

El resultado final del *Canvas* de simulación es el siguiente (figura 6):

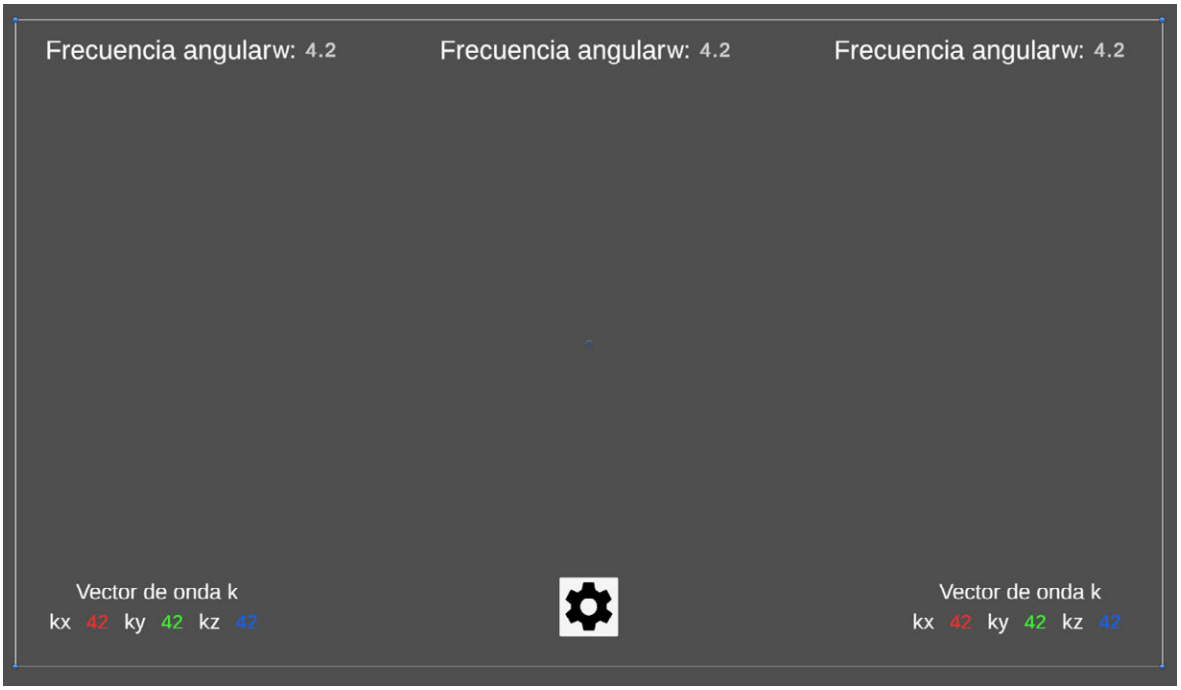

Figura 6: Resultado final del *Canvas* de simulación

4.9.2 Automatizando los textos de la UI

Como se puede observar en la figura 6, los textos de la UI, de momento, sólo tienen marcadores temporales. Se crea una lógica para automatizar el contenido de cada texto mediante el uso de *ScriptableObjects*.

ScriptableObject es una clase de Unity que permite almacenar grandes cantidades de datos compartidos independientes de las instancias de los *scripts*. Usar *ScriptableObjects* facilita el manejo de cambios y la automatización de múltiples procesos. También permite construir un nivel de comunicación flexible entre diferentes sistemas de la aplicación.

Se crea una clase *InputFieldDataSO*, que hereda de *ScriptableObject* (por convenio, todas las clases que heredan de *ScriptableObject* terminan su nombre con “SO”).

Se crea un *enum* que determina si el tipo de campo de entrada es *Vector3* o *float*.

Se crean atributos de tipo *string* para el título y los textos de x, y, z.

Se crea un método para leer el título.

Se crea un método para leer los textos, con dos sobrecargas. La primera sólo devuelve el texto de x, y va a ser el utilizado por los campos de entrada de tipo *float*. La segunda devuelve un array de *string* con los textos de los tres ejes y va a ser utilizado por los campos de entrada de tipo *Vector3*.

Una vez creada la clase, se crean instancias de dicha clase en forma de *Assets* de tipo *ScriptableObject*. Se crea uno para cada parámetro (amplitud, frecuencia angular, fase inicial, tamaño de rejilla y vector de onda $\vec{k}$) y se rellenan los datos como se ilustra en la figura 7.

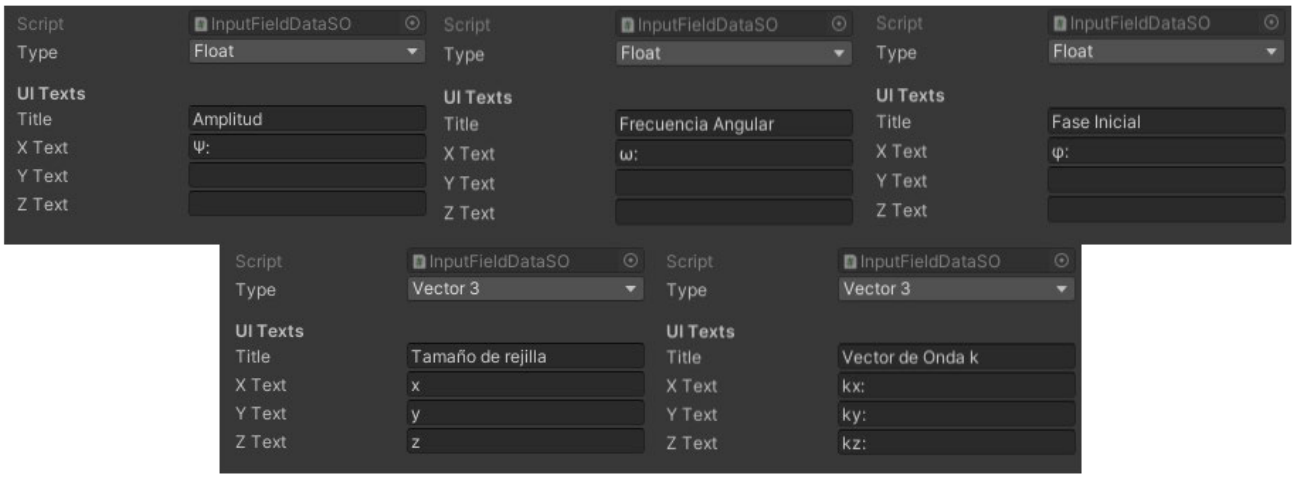

Figura 7: Configuración de los *ScriptableObjects* de los campos de entrada

Una vez creados los datos, éstos han de ser asociados con los textos de los campos de entrada. Para ello se crea una clase *InputFieldData* que hereda de *MonoBehaviour* y se agrega como componente a los objetos prefabricados de los campos de entrada (al hacer modificaciones en los objetos prefabricados se aplican dichos cambios en todas las instancias de dichos objetos en la jerarquía de la escena).

Se crea un atributo de tipo *InputFieldDataSO*, que se asocia al *Asset* de tipo *ScriptableObject* con los datos correspondientes a cada campo de entrada en el editor de Unity.

Se crean cuatro atributos de tipo *TextMeshProUGUI* que van a estar asociados a los cuatro posibles campos de texto del prefabricado del campo de entrada (en el prefabricado del campo de entrada *float*, los textos de y, z no se usan).

En la función *Start* de *MonoBehaviour* se ejecuta la siguiente lógica:

```
titulo.texto = xTexto.texto = textos[0].LeerTitulo
Si inputFieldDataSO.tipo es float
     xTexto.texto = inputFieldDataSO.LeerTexto
Si no
     string[] textos = inputFieldDataSO.LeerTexto
     xTexto.texto = textos[0]
     yTexto.texto = textos[1]
     zTexto.texto = textos[2]
```

De esta manera, si el campo de entrada es de tipo *float*, se usa la sobrecarga del método LeerTexto de *inputFieldDataSO* que devuelve un *string* y sólo se rellena un campo de texto. Si el campo de entrada es de tipo *Vector3*, se usa la sobrecarga del método LeerTexto de *inputFieldDataSO* que devuelve un array de tipo *string* y se rellenan los tres campos de texto.

En la figura 8 se observa el resultado final, visible cuando se inicia la aplicación:

Figura 8: *Canvas* de simulación con textos automatizados

4.9.3 Lógica del botón de configuración

Cuando se activa o desactiva un *GameObject* en la jerarquía de la escena de Unity, todos sus hijos se activan o desactivan con él. El botón de configuración se tiene que encargar de activar y desactivar el panel de configuración de la onda.

Para ello se crea una clase *ToggleGameObject* que hereda de *MonoBehaviour* y se pone como componente en el panel de configuración de la onda.

Se crea un método que, si el *GameObject* del que es componente está activo, lo desactiva y viceversa.

Como se puede observar en la figura 9, se asocia el método del componente *ToggleGameObject* del panel de configuración de la onda al evento de Unity *OnClick* del botón.

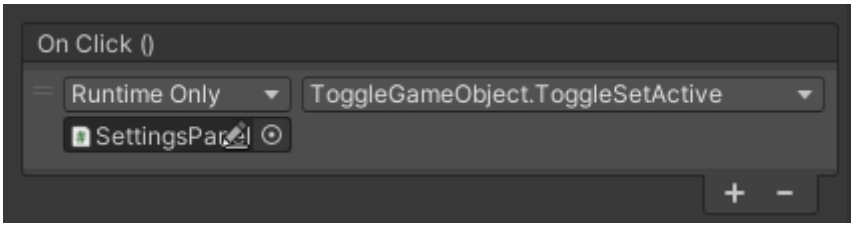

Figura 9: Evento de Unity *OnClick* del botón de configuración

Ahora, cada vez que el usuario clique el botón de configuración, se activa y desactiva el panel de configuración de la onda.

4.9.4 Lógica de los campos de entrada

Muchos de los parámetros de la onda guardan relación con otros parámetros. Esto implica que, si el usuario modifica un parámetro, todos aquellos relacionados con dicho parámetro han de actualizarse debidamente. Para ello, cada uno de los campos de entrada debe poder diferenciarse entre los demás y modificar los parámetros con los que guarda relación en consecuencia cuando es modificado.

Se crea una clase *InputFieldValue* que hereda de *MonoBehaviour* y se adjunta como componente en los campos de entrada de los objetos prefabricados.

Se crea un *enum* que determina el tipo de dato que contiene el campo de entrada.

Se crea una instancia de dicho *enum* para modificarla desde el editor de Unity.

Se crea un atributo de tipo *TMP_InputField* que referencia al propio campo de entrada, para poder modificar su texto cuando otro campo de entrada con relación al mismo se modifique.

En la función *Start* de *MonoBehaviour* se realiza una suscripción a la acción OndaActualizada de la instancia estática de la clase *ParticleManager*, que se asocia a un método, todavía por crear, ActualizarTexto. Dicho método se ejecuta también en *Start.*

Se crea un método para actualizar el texto del campo de entrada. En función del tipo de dato del campo de entrada, definido por el *enum* creado anteriormente, se recoge el valor correspondiente de la instancia estática de la clase *ParticleManager*, se convierte a *string*, se formatea para que sólo aparezca con un decimal, se añade el carácter de Pi donde es necesario (en la frecuencia angular y en las componentes del vector de onda $\vec{k}$) y se asigna al texto del campo de entrada.

Se crea un método de actualización de campo, que se asigna al evento del componente de campo de entrada que se ejecuta cada vez que el usuario cambia el contenido de este. En función del tipo de dato del campo de entrada, definido por el *enum* creado anteriormente, se recoge el valor introducido por el usuario, se modifica el atributo correspondiente y se actualizan todos aquellos que guardan relación con él en la instancia estática de *ParticleManager*.

Ahora, cada vez que el usuario modifique un parámetro de la onda, todos los parámetros que guardan relación se modifican también y se actualizan todos los textos de los campos de entrada, como se puede observar en la figura 10.

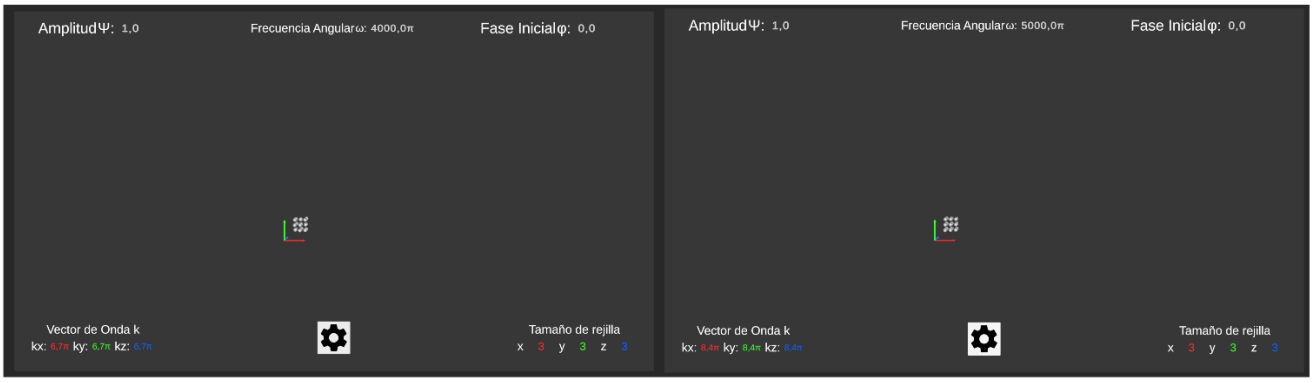

Figura 10: Cambio automático en el vector de onda $\vec{k}$ al cambiar la frecuencia angular

4.9.5 El Canvas del menú principal

Se crea un *Canvas* destinado a ser la UI del menú principal.

Se crea un texto en la parte superior con el nombre de la aplicación.

Se crea un panel con una imagen translúcida de fondo.

Dentro de dicho panel se crean dos botones. Uno para iniciar la simulación y otro para iniciar la simulación con tutorial.

Se crea un botón para salir de la aplicación.

Se distribuyen en el *Canvas* como se observa en la figura 11:

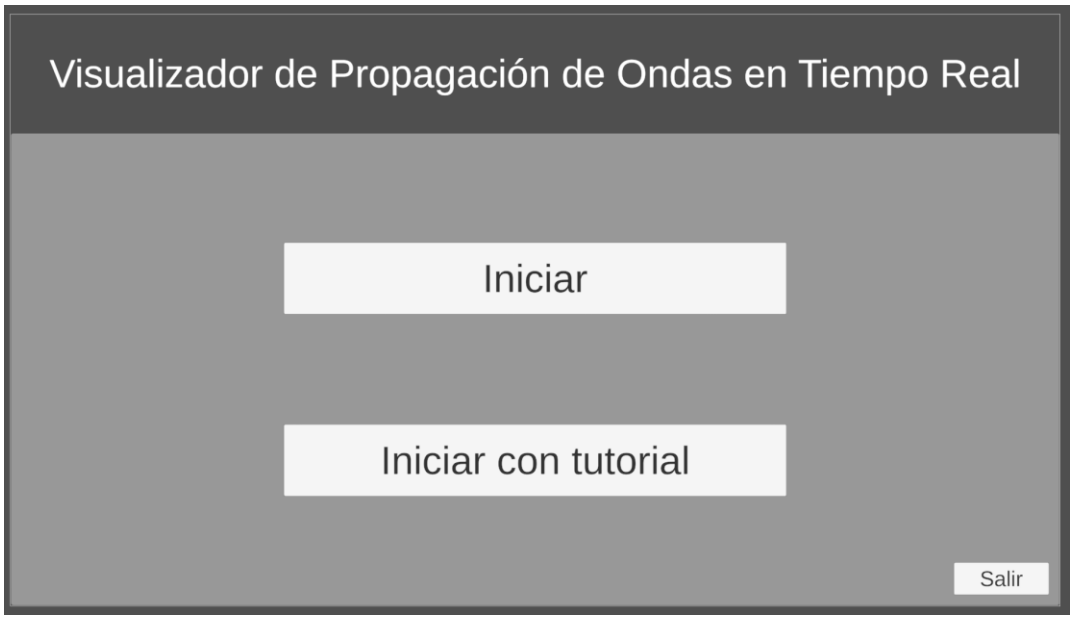

Figura 11: Distribución final del *Canvas del* menú principal

4.9.6 El Canvas del menú de pausa

El *Canvas* del menú de pausa es exactamente igual que el del menú principal, sólo que el botón de iniciar es para reanudar, tal como se observa en la figura 12:

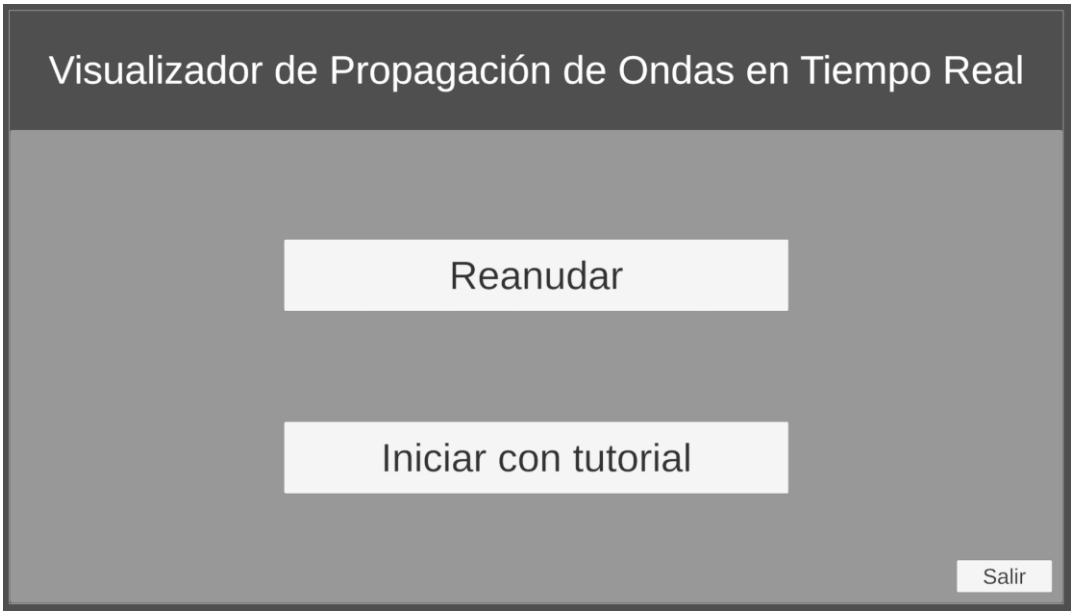

Figura 12: Distribución final del *Canvas* del menú de pausa

La lógica de los botones del menú se trata en la arquitectura de la aplicación.

4.10 La arquitectura

Se ha diseñado una arquitectura de máquina de estados

La arquitectura de máquina de estados es un patrón de diseño que permite modelar el comportamiento de un sistema que puede estar en diferentes estados y que cambia de estado según los eventos que recibe. Una máquina de estados se compone de los siguientes elementos:

- Estados: son las posibles situaciones en las que puede estar el sistema o el componente. Cada estado tiene asociado un conjunto de acciones que se ejecutan al entrar, al salir o al permanecer en el estado.
- Transiciones: son los cambios de estado que ocurren cuando se produce un evento. Cada transición tiene asociado una condición que determina si se puede realizar o no, y una acción que se ejecuta al realizarla.
- Eventos: son las señales o los estímulos que provocan las transiciones. Los eventos pueden ser externos (provenientes de otras clases) o internos (generados por el propio sistema o componente).

En la arquitectura diseñada los estados son los siguientes:

- Inicio
- Pausa
- Simulación
- Tutorial

Las transiciones posibles se pueden visualizar en la figura 13:

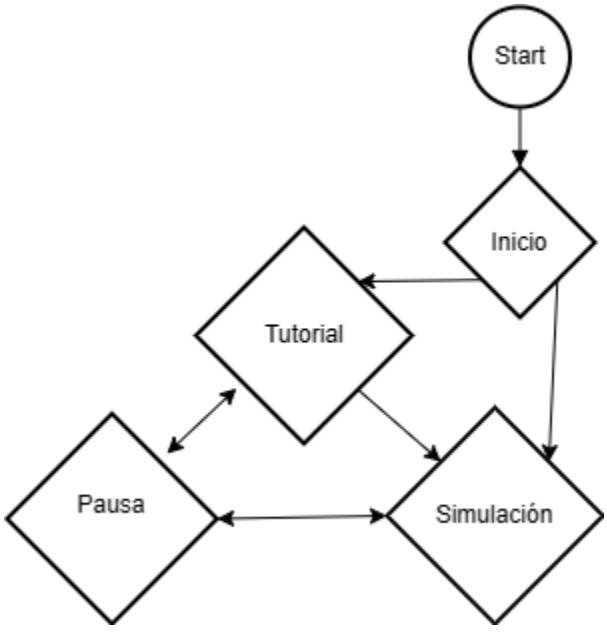

Figura 13: Diagrama de la máquina de estados

Los eventos que activan las transiciones son los siguientes:

- Inicio → Simulación: Botón “Iniciar” del menú principal
- Inicio → Tutorial: Botón “Iniciar con tutorial” del menú principal
- Tutorial → Simulación: Último botón de “aceptar” del tutorial
- Simulación → Pausa: Tecla *Escape*
- Pausa → Simulación: Tecla *Escape* o botón “Reanudar” del menú de pausa
- Pausa → Tutorial: Botón “Iniciar con tutorial” del menú de pausa

4.10.1 La clase abstracta *AppState*

Todos los estados comparten los siguientes métodos:

- Construir
- ActualizarEstado
- Destruir

Esto permite poder encapsular el código de manera eficiente. Hay varias maneras posibles y válidas de hacer esto.

Se puede usar una interfaz de la que hereden los estados, que los obligue a implementar las funciones que tienen en común

Se puede usar una clase abstracta de la que hereden los estados y en dicha clase crear los métodos virtuales correspondientes, para que cada estado pueda sobreescribirla.

En este caso se ha elegido la segunda opción.

Se crea la clase abstracta *AppState*, que hereda de *MonoBehaviour*.

Se crean los métodos virtuales Construir, ActualizarEstado y Destruir.

Con esto ya está todo listo para empezar a desarrollar la lógica de cada estado.

4.10.2 El estado de inicio

En el estado de inicio debe estar activo el *Canvas* del menú principal. A su vez, el estado de inicio es el que gestiona las funciones de los botones de inicio e inicio con tutorial del menú principal.

Se crea una clase *AppStateInit* que hereda de *AppState*.

Se crea un atributo de tipo *GameObject* para referenciar al *Canvas* del menú principal desde el editor de Unity.

Se crea la sobreescritura del método de construcción de *AppState*, que en el estado de inicio activa el *GameObject* del *Canvas* del menú principal.

Se crea la sobreescritura del método de destrucción de *AppState*, que en el estado de inicio desactiva el *GameObject* del *Canvas* del menú principal.

Se crea un método *OnPlayClick*, que cambia al estado de simulación.

Se crea un método *OnTutorialClick*, que cambia al estado de tutorial.

Se asignan dichos métodos a los botones del menú, como se observa en la figura 14:

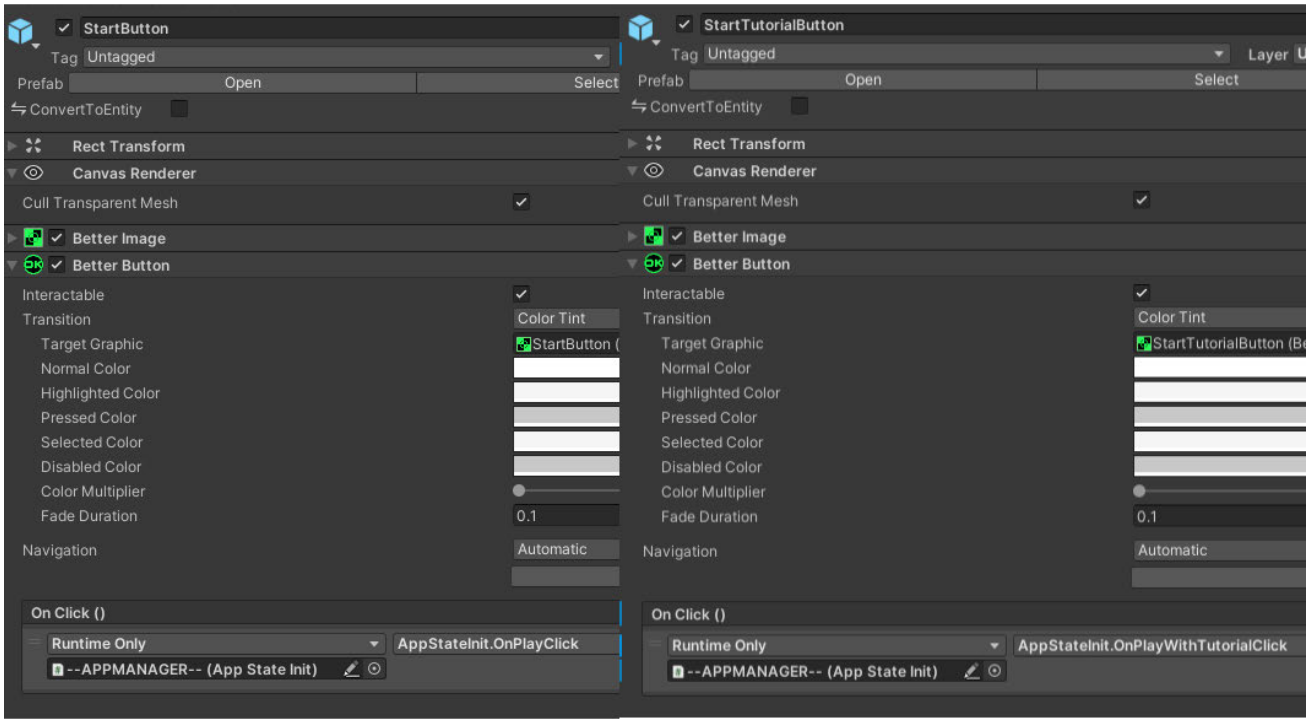

Figura 14: Asignación de botones *AppStateInit*

4.10.3 El estado de pausa

En el estado de pausa debe estar activo el *Canvas* del menú de pausa. A su vez, el estado de pausa es el que gestiona las funciones de los botones de reanudar e inicio con tutorial del menú de pausa.

Se crea una clase *AppStatePause* que hereda de *AppState*.

Se crea un atributo de tipo *GameObject* para referenciar al *Canvas* del menú de pausa desde el editor de Unity.

Se crea la sobreescritura del método de construcción de *AppState*, que en el estado de pausa activa el *GameObject* del *Canvas* del menú de pausa.

Se crea la sobreescritura del método de destrucción de *AppState*, que en el estado de pausa desactiva el *GameObject* del *Canvas* del menú de pausa.

Se crea un método *OnPlayClick*, que cambia al estado de simulación.

Se crea un método *OnPlayWithTutorialClick*, que cambia al estado de tutorial.

Se asignan dichos métodos a los botones del menú, igual que en el estado de inicio.

4.10.4 El estado de simulación

En el estado de simulación debe estar activo el *Canvas* del menú de simulación.

Se crea una clase *AppStateSimulation* que hereda de *AppState*.

Se crea un atributo de tipo *GameObject* para referenciar al *Canvas* del menú de simulación desde el editor de Unity.

Se crea un atributo de tipo *ParticleSpwaner* para referenciar el generador de partículas de la escena desde el editor de Unity.

Se crea la sobreescritura del método de construcción de *AppState*, que en el estado de simulación activa el *GameObject* del *Canvas* del menú de simulación, inicializa la rejilla desde el generador de partículas, activa la suscripción a la acción del *InputSystem* de pausa realizada, activa el mapa de acciones de jugador y desactiva el mapa de acciones de interfaz de usuario en el *InputSystem*.

Se crea la sobreescritura del método de destrucción de *AppState*, que en el estado de simulación desactiva el *GameObject* del *Canvas* del menú de simulación, desactiva la suscripción a la acción del *InputSystem* de pausa realizada, desactiva el mapa de acciones de jugador y activa el mapa de acciones de interfaz de usuario en el *InputSystem*.

Se crea un método *PauseOnPerformed*, que se ejecuta cuando la acción de pausa realizada da la señal (se pulsa la tecla escape), cambia al estado de pausa.

4.10.5 El estado de tutorial

La interfaz de usuario del tutorial está contenida en un panel dentro del *Canvas* de simulación. La interacción del usuario con el tutorial se basa en dos tipos de paneles (ambos hijos del panel mencionado anteriormente). Un panel explicativo con instrucciones y un botón de aceptar, para ir al siguiente paso y otro panel con imágenes transparentes para bloquear la interacción del usuario con otras partes del programa que no sean la explicada por el primer panel. En secciones futuras se entra más en detalle sobre esta mecánica.

En el estado de tutorial deben estar activos el *Canvas* del menú de simulación y el panel del tutorial. A su vez, el estado de tutorial es el que gestiona las funciones de los botones de reanudar e inicio con tutorial del menú de pausa.

Se crea una clase *AppStateTutorial* que hereda de *AppState*.

Se crean dos atributos de tipo *GameObject* para referenciar al *Canvas* del menú de simulación y al panel del tutorial desde el editor de Unity.

Se crean dos *arrays* de tipo *GameObject* para referenciar a los paneles, tanto de explicación como de bloqueo del tutorial desde el editor de Unity.

Se crea un atributo privado de tipo *int* como índice a la hora de navegar por los *arrays* de los paneles.

Se crea la sobreescritura del método de construcción de *AppState*, que en el estado de tutorial activa los *GameObject* del *Canvas* del menú de simulación y el panel del tutorial, inicializa el índice de navegación de los *arrays* a cero y activa los primeros paneles de explicación y bloqueo del tutorial.

Se crea la sobreescritura del método de destrucción de *AppState*, que en el estado de tutorial desactiva los *GameObject* del *Canvas* del menú de simulación y el panel del tutorial.

Se crea un método *TogglePanels*, que invierte el estado de activación de los paneles de explicación y bloque del tutorial del número de índice de los *arrays* por el que esté el índice de navegación.

Se crea un método *OnAcceptButtonClick*, que desactiva los paneles actuales, suma uno al índice de navegación y, si el índice de navegación es menor a la longitud de los *arrays*, activa los paneles siguientes. Si no, cambia al estado de simulación.

Se asigna dicho método al botón de aceptar de los paneles explicativos.

4.10.6 El Manejador de la aplicación

Para manejar en qué estado se encuentra la aplicación en cada momento se crea una clase *AppManager*, que hereda de *MonoBehaviour*. Tanto *AppManager*, como las clases que heredan de *AppState* se colocan en el mismo *GameObject* como componentes.

La clase *AppManager* es un manejador, por lo que se realiza la misma lógica que en la clase *ParticleManager* para que tenga una única instancia estática, añadiendo un atributo estático de su propia clase con el nombre *Instance*.

Se crea un atributo privado de tipo *AppState*, capaz de albergar un objeto de cualquier clase que herede de *AppState*, que actúa como estado actual.

En el método *Awake* de *MonoBehaviour* se establece el estado inicial como estado y se invoca su método de construcción.

En el método *Update* de *MonoBehaviour* se invoca el método de actualización de estado del estado actual.

Se crea un método *ChangeState* que invoca el método de destrucción del estado actual, establece el nuevo estado como estado actual e invoca el método de construcción del nuevo estado.

Se crea un método *ExitApp* que se asigna a los botones de salir del *Canvas* de inicio y del *Canvas* de pausa, que cierra la aplicación en el sistema operativo correspondiente.

4.11 El Tutorial

Como ya se ha mencionado, la interfaz de usuario del tutorial está contenida en un panel dentro del *Canvas* de simulación. La interacción del usuario con el tutorial se basa en dos tipos de paneles (ambos hijos del panel mencionado anteriormente). Un panel explicativo con instrucciones y un botón de aceptar, para ir al siguiente paso y otro panel con imágenes transparentes para bloquear la interacción del usuario con otras partes del programa que no sean la explicada por el primer panel.

4.11.1 El panel explicativo

Se crea un panel con un componente de imagen. Dentro de dicho panel se crea un texto destinado a contener la explicación correspondiente y un botón de aceptar. Se centra el panel en el *Canvas* y se ajustan los tamaños del texto y del botón. Se observa el resultado en la figura 15:

Figura 15: Maquetación del panel explicativo del tutorial

El relleno del texto explicativo se realiza automáticamente mediante *ScriptableObjects*.

Para ello se crea una clase *TutorialExplanationInfoSO*, que hereda de *ScriptableObject*.

Se añade un atributo público de tipo *string* que va a contener el texto de la explicación. Se configura dicho atributo para que en el editor de Unity aparezca como un área de texto.

Se crea una clase *TutorialExplanationData*, que hereda de *MonoBehaviour* y se añade como componente al panel explicativo.

Se añade un atributo de tipo *TutorialExplanationInfoSO*, que va a contener la referencia al *ScriptableObject* correspondiente desde el editor de Unity.

Se añade un atributo de tipo *TextMeshProUGUI* que va a contener la referencia al propio texto del panel desde el editor de Unity

En el método *Start* de *MonoBehaviour* se asigna al texto del panel el contenido del *ScriptableObject*.

Se crean los *ScriptableObjects* y se rellenan con las explicaciones correspondientes a cada paso del tutorial.

4.11.2 Los paneles de bloqueo

Se crean paneles con combinaciones imágenes translúcidas para bloquear la interacción del usuario con las partes de la interfaz que no tengan que ver con la sección del tutorial en la que se encuentre.

En algunos casos se añaden dos imágenes de flechas para indicar con qué zona ha de interactuar el usuario en esa fase del tutorial.

En la figura 16 se observan los 7 paneles resultantes:

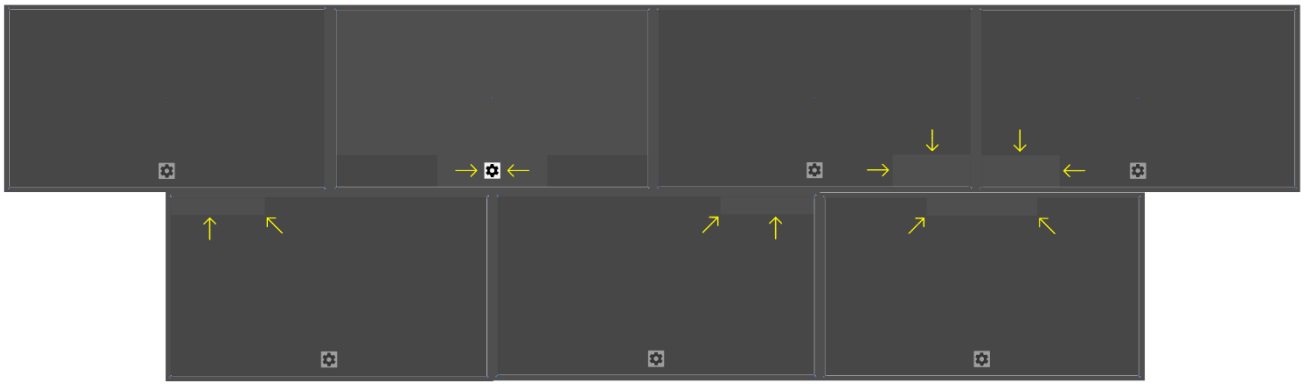

Figura 16: Paneles de bloque de interacción del tutorial

4.11.3 Implementando los paneles en el componente *AppStateTutorial*

Se implementan los paneles en los *arrays* destinados a ellos mencionados en la sección 4.10.5.

Para ello se han de arrastrar los paneles creados a su correspondiente array en orden. Dado que algunos paneles explicativos comparten paneles de bloqueo, éstos últimos se repiten en el *array*, de ser necesario. En la figura 17 se puede observar el orden final:

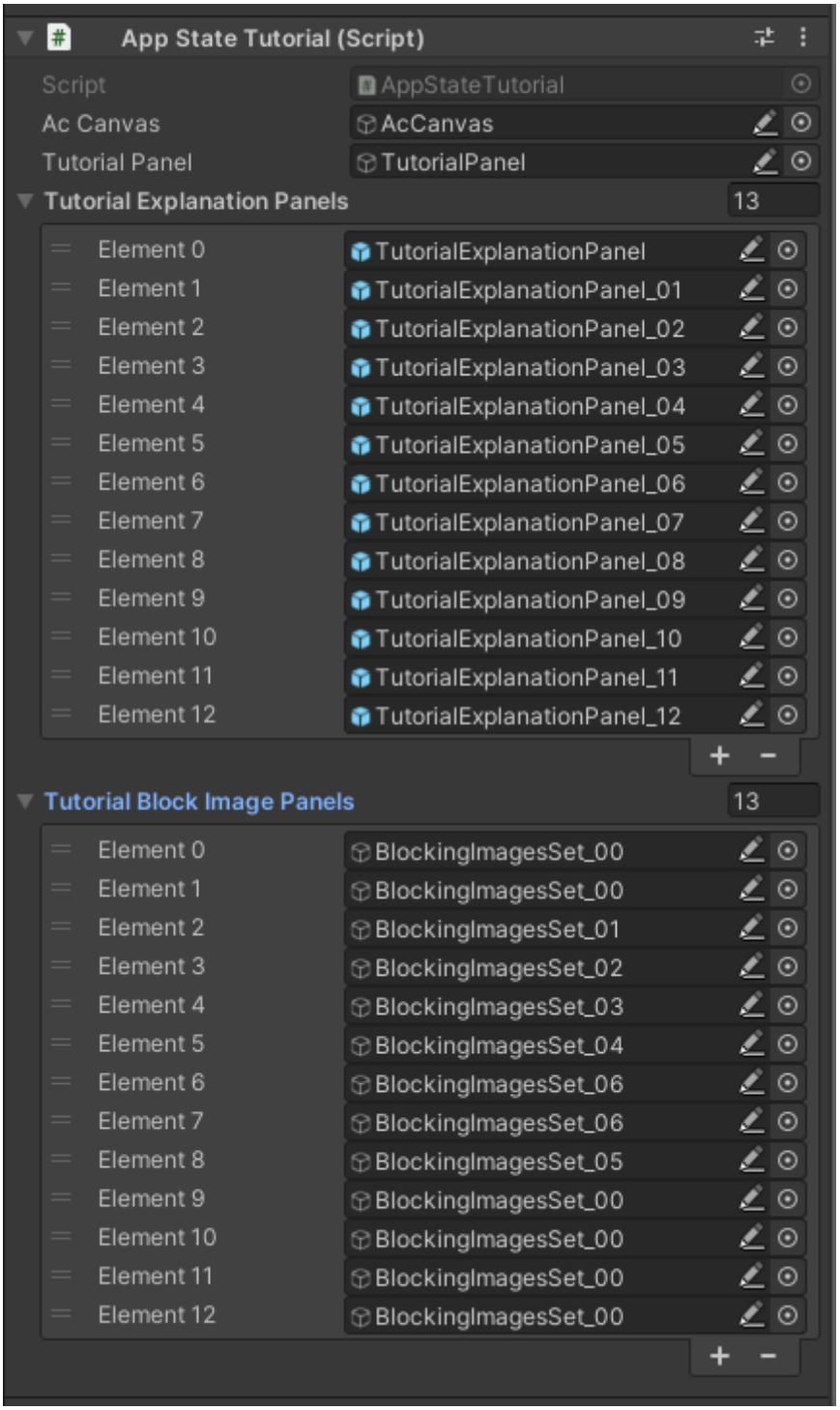

Figura 17: Orden de paneles explicativos y de bloqueo en el estado de tutorial

5 Resultados

El parámetro más interesante para medir los resultados de este proyecto es la diferencia de rendimiento entre los flujos de trabajos con *GameObjects* y con el sistema DOTS. Para ello se va a medir la media de FPS, que va ligada directamente con el tiempo de procesado y, por lo tanto, permite comprobar más rápidamente si existe un cuello de botella de rendimiento. A su vez se va a medir el estrés al que es sometido la tarjeta gráfica mediante los *batches* o carga de llamadas a la tarjeta gráfica, que es otra fuente muy común de cuellos de botella de rendimiento.

Se comparan los resultados de rendimiento con los flujos de trabajo de *GameObjects* y el sistema DOTS en pantalla completa, con una resolución de 1920x1080 píxeles en un ordenador portátil con procesador 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz, 32GB de memoria RAM, tarjeta gráfica *Nvidia Geforce* RTX 3080 16GB y Windows 10 Pro de 64 bits en tres casos diferentes:

- Con una sola partícula
- Con una rejilla de partículas de 16x16x16
- Con una rejilla de partículas de 32x32x32

5.1 Una partícula

De los resultados de la tabla 1 se pueden extraer varias conclusiones. Al haber sólo una partícula en pantalla, la diferencia de rendimiento es muy pequeña y, en magnitudes de 400 FPS, irrelevante, ya que los monitores de gama alta alcanzan los 360Hz de tasa de refresco. También se puede deducir que la interfaz de usuario y la malla de los ejes de referencia suponen 14 *batches*, 308 triángulos y 322 vértices, ya que una sola partícula consta de 2 triángulos, 4 vértices y supone una llamada a la tarjeta gráfica.

Tabla 1: Estadísticas de rendimiento con una partícula

	GameObjects	**Entidades**
Media de FPS	400	440
Tiempo de procesado	2.5ms	2.2ms
Batches	15	15
Triángulos	310	310
Vértices	326	326

5.2 Rejilla de partículas de 16x16x16

De los resultados de la tabla 2 se puede extraer que usando las entidades del sistema DOTS para renderizar el mismo número de triángulos y vértices la aplicación es, aproximadamente 10 veces más eficiente y reduce la carga de llamadas a la tarjeta gráfica (*batches*) 190 veces, aproximadamente.

Tabla 2: Estadísticas de rendimiento con rejilla de 16x16x16

	GameObjects	**Entidades**
Media de FPS	32	310
Tiempo de procesado	30ms	3.2ms
Batches	8201	43
Triángulos	16.6k	16.6k
Vértices	33.2k	33.2k

5.3 Rejilla de partículas de 32x32x32

De los resultados de la tabla 3 se puede extraer que usando las entidades del sistema DOTS para renderizar el mismo número de triángulos y vértices la aplicación es, aproximadamente 18,5 veces más eficiente y reduce la carga de llamadas a la tarjeta gráfica (*batches*) 649 veces.

Tabla 3: Estadísticas de rendimiento con rejilla de 32x32x32

	GameObjects	**Entidades**
Media de FPS	5.3	100
Tiempo de procesado	185ms	10ms
Batches	65549	101
Triángulos	131.4k	131.4k
Vértices	262.5k	262.5k

Estos datos permiten concluir que, cuanto mayor es el número de elementos en pantalla, la eficiencia del sistema DOTS con respecto a los *GameObjects* incrementa

exponencialmente. Un incremento de 8 veces la cantidad de elementos en pantalla supone el doble de eficiencia de procesado y 3.5 veces la eficiencia en la carga de llamadas a la tarjeta gráfica.

6 Planos

En este apartado se proporcionan los diagramas de clases que ilustran la relación entre los diferentes *scripts* de las principales estructuras de código de la aplicación.

6.1 Arquitectura

Como se observa en la figura 18, AppManager puede manejar en qué estado se encuentra la aplicación a través de su atributo *state* de tipo *AppState*. *AppState* es una clase abstracta, por lo que cualquier clase puede reescribir cualquiera de sus tres métodos. Eso quiere decir que el objeto de la clase de las herederas de *AppState* que esté guardado en el atributo *state* es la que aplica su propia versión de los métodos virtuales *Construct*, *Desctruct* y *UpdateState* de *AppState*, a los cuales se llama desde *AppManager*.

Tanto *AppManager*, como las clases herederas de *AppState*, al heredar de *MonoBehaviour*, pueden tener una instancia como componentes en un *GameObject* dentro de la escena de Unity.

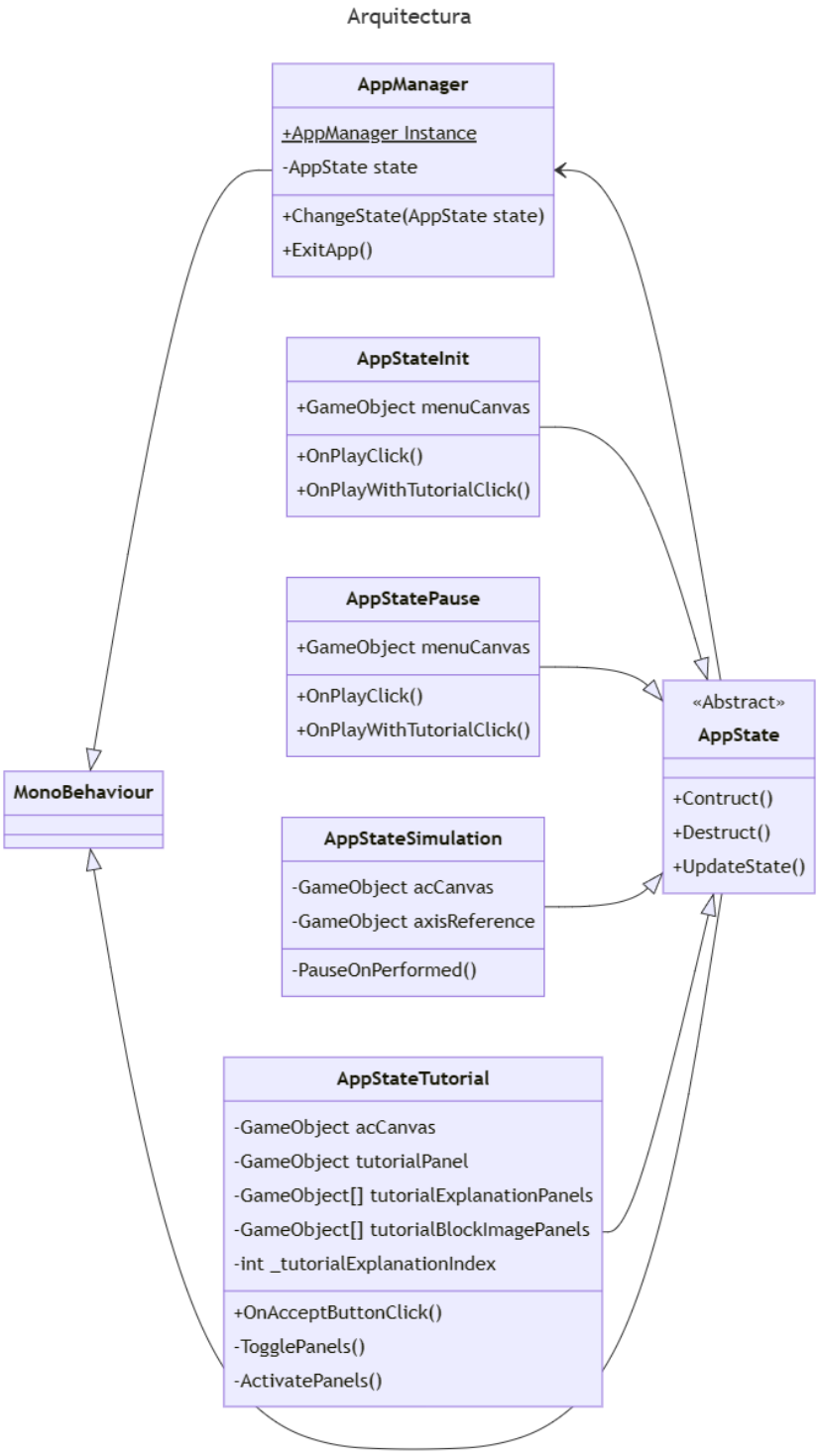

Figura 18: Diagrama de clases de la arquitectura de la aplicación

6.2 Manejo de entradas

Como se observa en la figura 19, la clase *InputManager* actúa como intérprete, implementando las acciones de hardware que llegan desde *Default_Input_Actions* y decidiendo qué clase de datos da cada una como salida. El resto de las clases se suscriben a las acciones de *InputManager* que necesitan. Dichas clases heredan de *MonoBehaviour* y cuentan con una instancia en un *GameObject* de la escena.

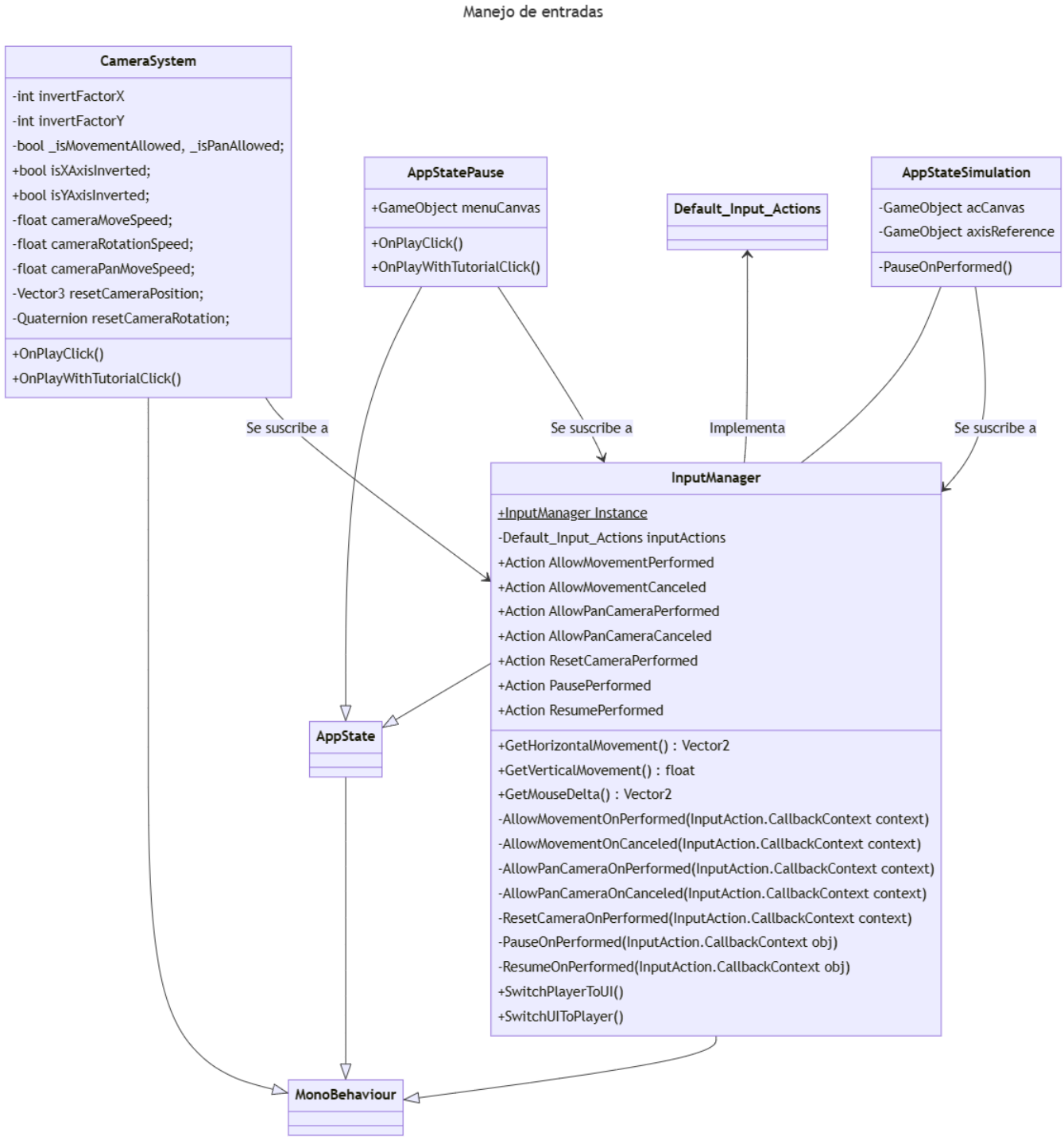

Figura 19: Diagrama de clases del manejo de entradas de la aplicación

6.3 Partículas

Como se observa en la figura 20, *SystemBase* e *IComponentData* son la clase abstracta e interfaz, respectivamente que hacen que las clases *AcousticWaveSystem* y *AcoustiWaveData* sean reconocidos como un sistema y un componente, respectivamente en la estructura ECS de DOTS. En la clase *ParticleSpawner*, el atributo *entityManager* de tipo *EntityManager* reconoce el componente y crea la rejilla de entidades del tamaño establecido y con el espaciado establecido. El sistema mueve todas las entidades que tienen el componente *AcousticWaveData*.

Mientras tanto, los datos del componente *AcousticWaveData* dependen de los atributos de *ParticleManager*. A su vez, la clase *InputFieldValue* está suscrita a la acción que invoca *ParticleManager* cada vez que se actualiza alguno de sus atributos, para modificar los textos de la interfaz de usuario de manera acorde.

InputFieldValue implementa el enumerador *DataType* para diferenciar de qué tipo es cada campo de datos en la interfaz de usuario.

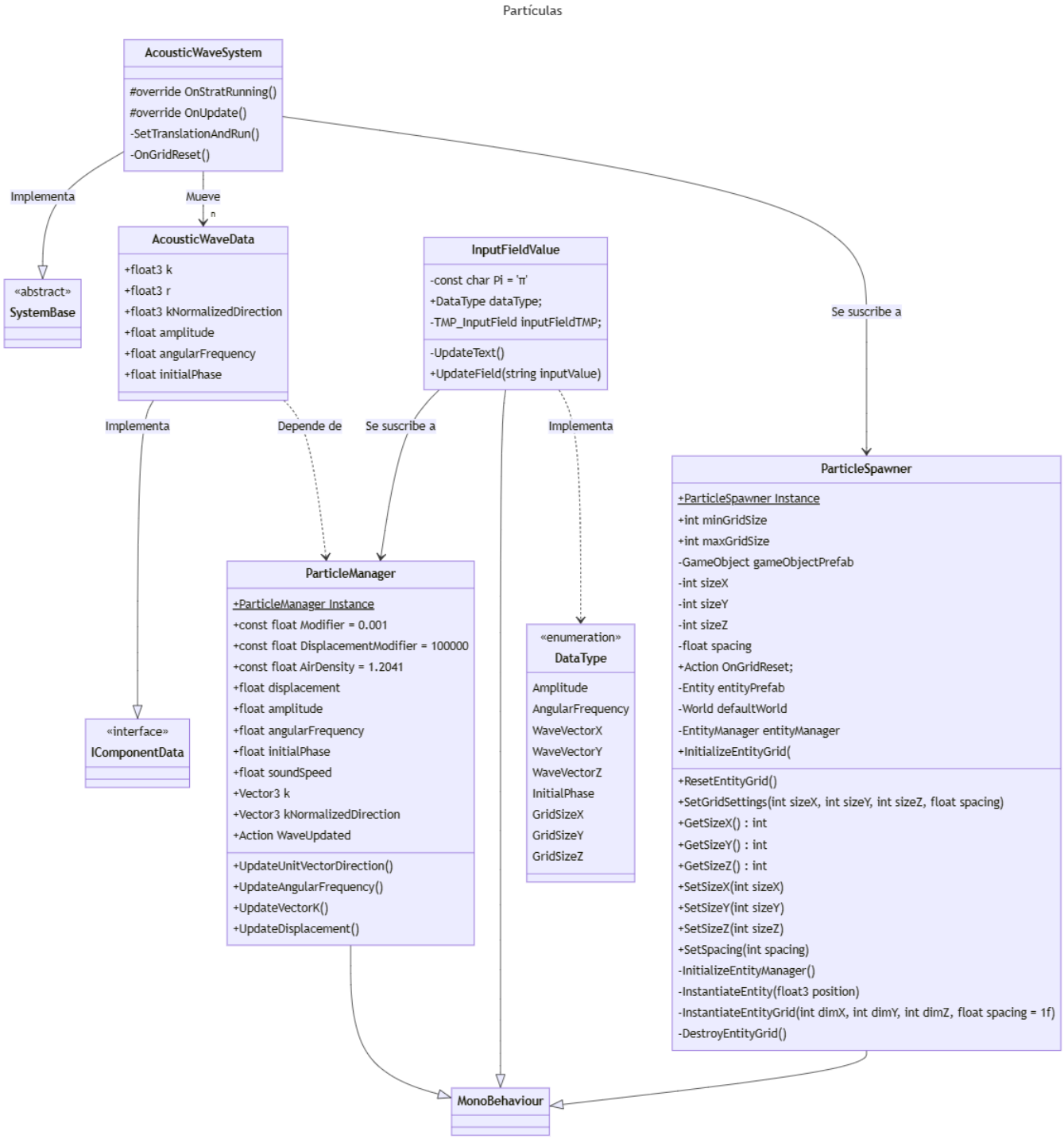

Figura 20: Diagrama de clases de la estructura de las partículas

7 Presupuesto

Para este proyecto se ha utilizado un ordenador portátil MSI GE66 Raider 11UH, la licencia gratuita de Unity, JetBrains Rider 2022 y el sueldo de un ingeniero durante 319 horas a lo largo de 3 meses:

- Ordenador portátil MSI GE66 Raider 11UH, con un precio de mercado de 3.499 euros. Se ha amortizado el 10% del valor del equipo, ya que se estima que se ha usado el 30% del tiempo durante los 3 meses que ha durado el proyecto.
- La licencia gratuita de Unity no tiene ningún coste asociado, pero se debe mencionar como herramienta utilizada.
- JetBrains Rider 2022 tiene un coste de 199 euros al año, pero se ha obtenido una licencia gratuita para estudiantes. Al igual que Unity, se debe mencionar como herramienta utilizada.
- El sueldo de un ingeniero de telecomunicaciones, que se ha estimado en 25 euros la hora. Se han dedicado 319 horas al proyecto, repartidas en 3 meses, lo que supone un coste total de 7.975 euros.

El presupuesto total del proyecto es de:

- Coste del ordenador portátil: 349,9 euros
- Coste de la licencia de JetBrains Rider 2022: 0 euros
- Coste del sueldo del ingeniero: 7.975 euros
- Coste total: 8.324,9 euros

8 Manual de usuario

8.1 Inicio de la aplicación

Para iniciar la aplicación, siga los siguientes pasos:

1. Descomprima la carpeta en formato .rar (figura 21):

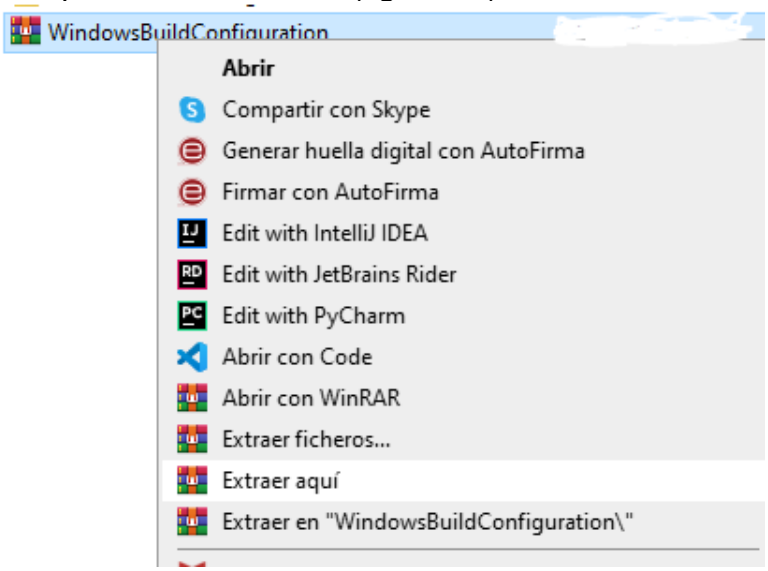

Figura 21: Descompresión de carpeta

2. Abra la carpeta de la aplicación (figura 22):

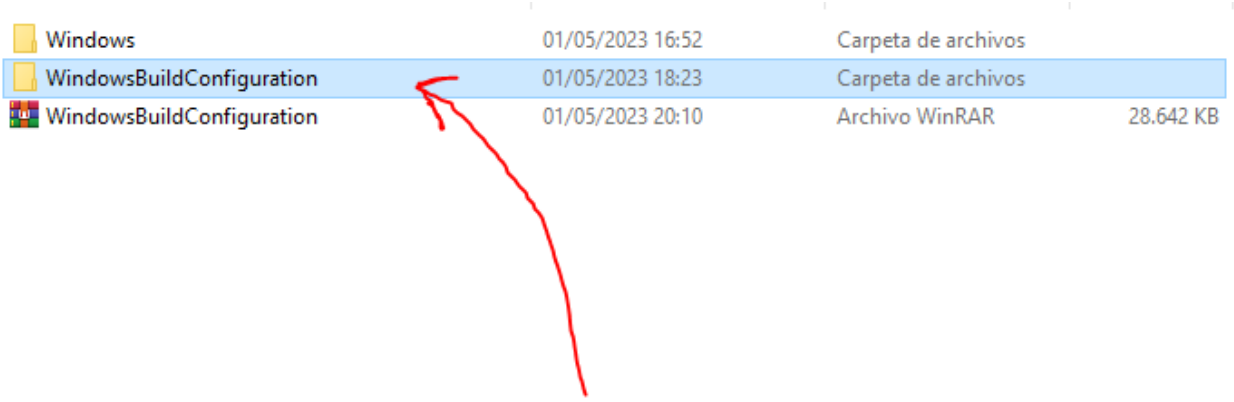

Figura 22: Apertura de la carpeta descomprimida

3. Haga doble clic sobre el ejecutable (figura 23):

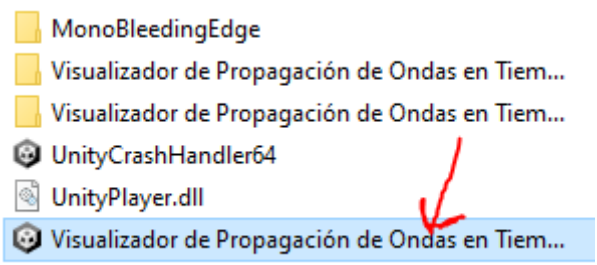

Figura 23: Ejecutable

8.2 Menú Principal

En el menú principal dispone de tres botones, con las siguientes funciones (figura 24):

1. Iniciar: Inicia la simulación
2. Iniciar con tutorial: Inicia la simulación con un tutorial previo
3. Salir: Cierra la aplicación

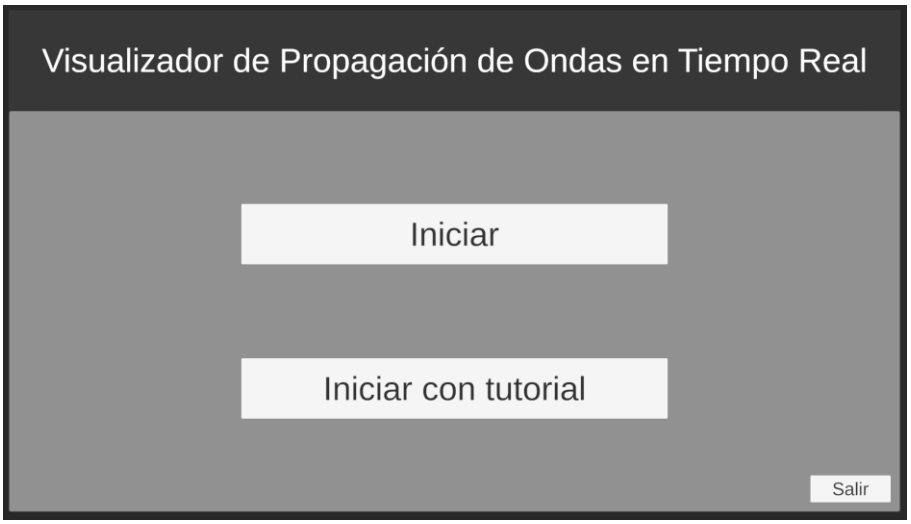

Figura 24: Menú principal

8.3 Simulación

La simulación comienza siempre con una rejilla de 3x3x3 partículas con los siguientes parámetros:

- Amplitud: 1
- Frecuencia angular: 4000π
- Vector de onda
  - X: 6.7π
  - Y: 6.7π
  - Z: 6.7π
- Fase inicial: 0

Para mostrar y ocultar los parámetros, clique sobre el engranaje (figura 25):

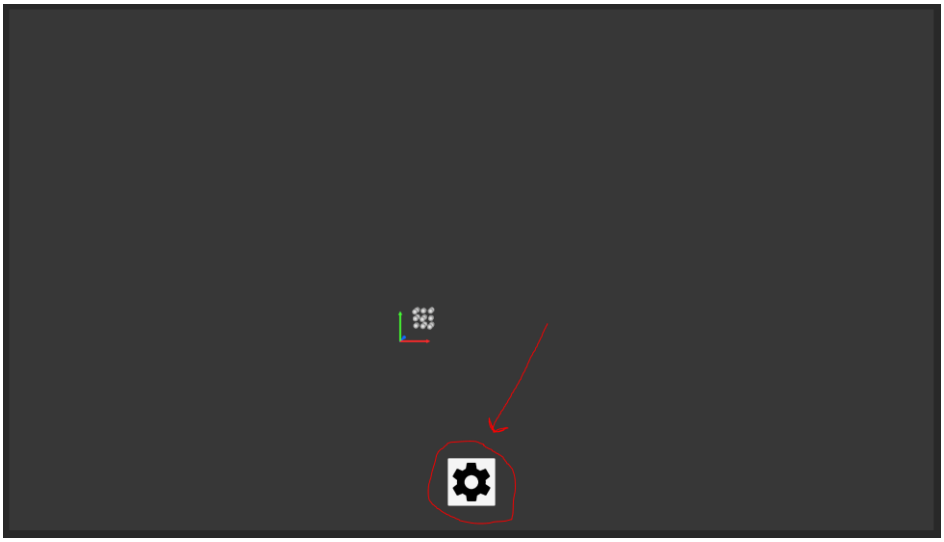

Figura 25: Botón de configuración

Para modificar cualquier parámetro, clique sobre el valor del parámetro, introduzca el valor deseado y presione “enter” o clique sobre cualquier otro punto de la pantalla (figura 26).

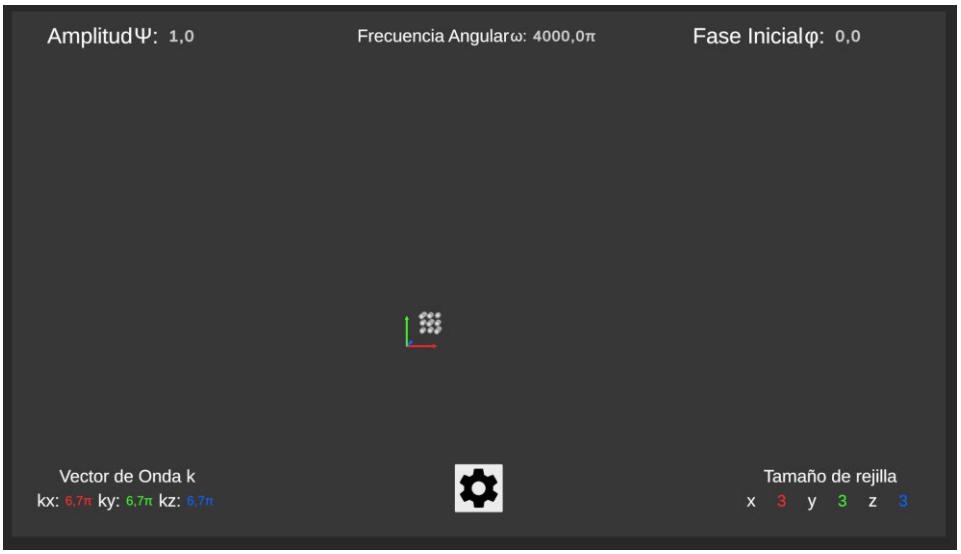

Figura 26: Parámetros de la onda

Cuestiones para tener en cuenta a la hora de cambiar los parámetros:

- El tamaño de la rejilla está limitado a 32 en cada eje para no forzar o dañar ningún equipo.
- El vector $\vec{k}$ está relacionado con la frecuencia angular ω por lo que, si modifica $\vec{k}$, ω se modificará de manera acorde.
- Al modificar la amplitud de la presión de la onda. Recuerde que lo que va a visualizar es el desplazamiento de la onda, por lo que la amplitud de la presión es un factor en la ecuación que relaciona dichas amplitudes.
- La frecuencia angular ω está relacionado con el vector $\vec{k}$ por lo que, si modifica ω, $\vec{k}$ se modificará de manera acorde.
- ω toma parte en la relación entre las amplitudes de presión y desplazamiento. Observará que, cuanto mayor sea ω, menor será el desplazamiento y viceversa.

- Al modificar la fase inicial, recuerde que el desplazamiento ya de por sí está retrasado $\pi/2$ con respecto a la presión.
- Para que la visualización sea posible, el valor final de ω ha sido modificado con un factor de 10^{-3} y el desplazamiento con un factor de 10^{5}.

Para mover y rotar la cámara puede mantener el clic derecho del ratón y moverla cámara en las siguientes direcciones:

- W → adelante
- S → atrás
- A → izquierda
- D → derecha
- Q → arriba
- E → abajo

Puede arrastrar la cámara manteniendo la rueda del ratón pulsada.

Puede reiniciar la posición y rotación de la cámara con la tecla F.

Puede usar la tecla escape para acceder al menú de pausa en cualquier momento.

8.4 Tutorial

El tutorial le guiará paso a paso por las opciones descritas en la sección 8.3, con descripciones detalladas de cada parámetro modificable.

Se ha de completar el tutorial para poder acceder al menú de pausa.

8.5 Menú de pausa

En el menú de pausa dispone de tres botones, con las siguientes funciones:

1. Reanudar: Reanuda la simulación
2. Iniciar con tutorial: Inicia la simulación con un tutorial previo
3. Salir: Cierra la aplicación

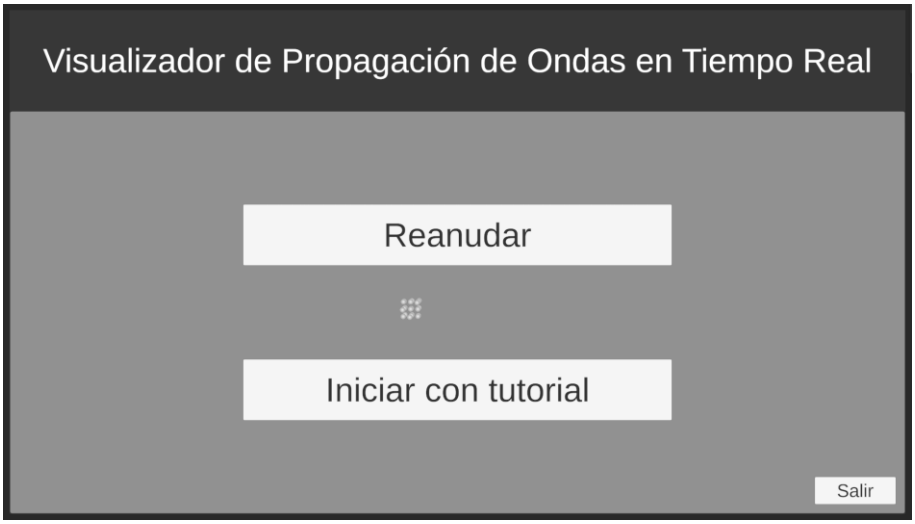

Figura 27: Menú de pausa

También puede reanudar la aplicación usando la tecla escape.

9 Impacto del proyecto

Este proyecto ha sido diseñado para servir como material de ayuda para la asignatura Propagación de Ondas, impartida en la Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunación.

El impacto de este proyecto se puede dividir en tres categorías:

- Impacto social
- Impacto económico
- Impacto medioambiental

9.1 Impacto Social

Teniendo como objetivo tanto ayudar a los alumnos a visualizar y entender mejor la teoría impartida en la asignatura de manera interactiva, como a los docentes a impartirla de manera más visual e intuitiva, el impacto social de este proyecto reside en la facilitación de la comunicación entre docente y alumnos mediante una herramienta visual e interactiva, que puede ser usada por el profesor o profesora para ejemplificar de manera sencilla cualquier concepto impartido, o puede ser usada por el alumno o alumna para experimentar en casa y entender mejor, de manera práctica, cualquier teoría que pueda ser de mayor dificultad.

9.2 Impacto económico

El impacto económico de este proyecto toma dos vertientes.

Por un lado, se ve afectado por el impacto social descrito anteriormente ya que, al mejorar la comunicación entre el docente y los alumnos, la tasa de aprobados de la asignatura se puede verse incrementada. Esto tiene como consecuencia una reducción en matrículas simultáneas en la asignatura y en matrículas consecutivas individuales para los alumnos, lo que se traduce, tanto en menor coste en materiales, aulas, tutorías individuales, etc., como en un valor mayor de la matrícula para el alumno, porque el profesor podrá dedicar un mayor tiempo a cada alumno de manera individual.

Por otro lado, el haber enfocado tanto esfuerzo en la eficiencia de la aplicación tiene como consecuencia un menor gasto de energía del equipo en el que se reproduzca y un deterioro más lento del mismo. Esto se traduce en un menor coste económico, tanto para los alumnos cuando usen la herramienta en su domicilio, como para las instituciones en las que la utilicen tomando prestado un equipo.

9.3 Impacto medioambiental

El impacto medioambiental del proyecto va ligado directamente con una de las vertientes del impacto económico:

Al haber maximizado la eficiencia de la aplicación y, en consecuencia, haber reducido notablemente su gasto de energía se produce una reducción en las emisiones que esa energía extra hubiera generado.

10 Conclusiones

El objetivo principal del proyecto era desarrollar una aplicación que permitiera visualizar y simular la propagación de ondas en diferentes medios y condiciones, utilizando para ello el motor gráfico Unity.

Se ha priorizado la eficiencia y robustez de la aplicación, teniendo que sacrificar la posibilidad de simular la propagación de ondas acústicas esféricas y electromagnéticas. Se puede cambiar el medio de propagación de manera sencilla desde el proyecto, pero finalmente al usuario sólo se le ha dado la posibilidad de visualizar las ondas en el aire. Los notables resultados de mejora de eficiencia justifican los objetivos no conseguidos con respecto al anteproyecto.

Se ha logrado realizar la simulación de ondas acústicas planas en el aire, pudiendo modificar los parámetros de la onda en tiempo real y con un espacio de simulación muy amplio. Se ha hecho especial énfasis en la escalabilidad de la aplicación, siendo muy sencillo ampliar el tipo de ondas y condiciones a simular.

Las conclusiones que se pueden extraer tras la realización del proyecto son las siguientes:

- Se ha demostrado que Unity es mucho más que un motor de videojuegos y que se puede utilizar para otros fines. Unity ofrece una serie de ventajas como la facilidad de uso, la portabilidad, la integración con otras herramientas y la calidad gráfica, que lo convierten en una opción muy interesante para desarrollar aplicaciones científicas y educativas.
- Se ha comprobado que el sistema DOTS de Unity es extremadamente útil para manejar cantidades masivas de objetos en pantalla con comportamientos parecidos. El sistema DOTS permite optimizar el rendimiento y el uso de recursos, aprovechando el paralelismo y la arquitectura orientada a datos. Gracias a este sistema, se ha podido simular la propagación de miles de ondas en tiempo real, con un nivel de detalle y realismo muy alto.

Una línea futura de trabajo es incluir en la aplicación, usando el esquema establecido en este proyecto como base, el resto de las simulaciones inicialmente previstas en el anteproyecto: ondas acústicas esféricas y ondas electromagnéticas.

Otras líneas de trabajo futuras pueden ser incluir opciones de selección de partículas, cambio de medio de propagación y adaptaciones visuales, como, por ejemplo, el cambio de color de las partículas en función de la velocidad de estas.

11 Referencias

1. **Los profesores de la asignatura.** Apuntes de Propagación de Ondas. *2-Ondas Acústicas Planas.* [En línea] [Citado el: 10 de Julio de 2023.]

2. **Haas, John K.** A History of the Unity Game Engine. *Digital WPI.* [En línea] 6 de Marzo de 2014. [Citado el: 2 de Octubre de 2022.] https://www.google.es/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwiCu72R8OH6AhUJ4oUKHdCyDlYQFnoECA4QAQ&url=https%3A%2F%2Fdigital.wpi.edu%2Fdownloads%2F2f75r821k&usg=AOvVaw0PLR1E2cqfaOEaRviN029I.

3. **Unity Technologies.** Página oficial de Unity. [En línea] [Citado el: 2 de Octubre de 2022.] https://unity.com/es.

4. **Cabanes, Nacho.** Introducción a la programación con C#. *Programación - Por Nacho Cabanes.* [En línea] 30 de Diciembre de 2012. [Citado el: 2 de Octubre de 2022.] https://www.nachocabanes.com/csharp/.

5. **Unity Technologies.** Manual de Unity. *Render Pipelines.* [En línea] 13 de Junio de 2023. https://docs.unity3d.com/2021.3/Documentation/Manual/render-pipelines.html.

6. **Unity Technologies.** Manual de Unity. *Input Manager.* [En línea] [Citado el: 13 de Junio de 2023.] https://docs.unity3d.com/2021.3/Documentation/Manual/class-InputManager.html.

7. **Unity Technologies.** Manual de Unity. *Input System.* [En línea] [Citado el: 13 de Junio de 2023.] https://docs.unity3d.com/Packages/com.unity.inputsystem@1.6/manual/index.html.

8. **Unity Technologies.** Página de introdución de DOTS. [En línea] [Citado el: 10 de Junio de 2023.] https://unity.com/dots.

9. **Unity Technologies.** Manual de Unity. *Entities 0.51.* [En línea] [Citado el: 10 de Junio de 2023.] https://docs.unity3d.com/Packages/com.unity.entities@0.51/manual/index.html.

10. **Unity Technologies.** Manual de Unity. *Job System.* [En línea] [Citado el: 10 de Junio de 2023.] https://docs.unity3d.com/Manual/JobSystem.html.

11. **Kinsler, y otros.** *Fundamentos de Acústica.* s.l. : Limus.

12. **Unity Technologies.** Unity Scripting API. *MonoBehaviour.* [En línea] [Citado el: 08 de Junio de 2023.] https://docs.unity3d.com/2021.3/Documentation/ScriptReference/MonoBehaviour.html.

13. **Unity Technologies.** Unity Scripting API. *GameObject.* [En línea] [Citado el: 08 de Junio de 2023.]

14. **Unity Technologies.** Unity Scripting API. *Home Page.* [En línea] [Citado el: 08 de Junio de 2023.] https://docs.unity3d.com/2021.3/Documentation/ScriptReference/index.html.

15. **Unity Technologies.** Manual de Unity. *Meshes.* [En línea] [Citado el: 13 de Junio de 2023.] https://docs.unity3d.com/2021.3/Documentation/Manual/mesh.html.

16. **Unity Technologies.** Manual de Unity. *Shaders.* [En línea] [Citado el: 13 de Junio de 2023.] https://docs.unity3d.com/2021.3/Documentation/Manual/Shaders.html.

17. **Unity Technologies.** Manual de Unity. *Materials.* [En línea] [Citado el: 13 de Junio de 2023.] https://docs.unity3d.com/2021.3/Documentation/Manual/Materials.html.

18. **Unity Technologies.** Unity Manual. *Mathematics.* [En línea] [Citado el: 10 de Junio de 2023.] https://docs.unity3d.com/Packages/com.unity.mathematics@1.2/manual/index.html.